SOFTWARE AND HARDWARE SUPPORT FOR DATA INTENSIVE COMPUTING

BY

MINGLIANG WEI

B.S., Nanjing University, Nanjing, China, 1998
M.E., Nanjing University, Nanjing, China, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

Many data-intensive applications exhibit poor temporal and spatial locality and perform poorly on commodity processors, due to high cache miss rates. Some due to unsophisticated implementations that do not exploit hardware potentials, others due to their inborn nature of poor data access locality. We address this problem by both software and hardware approaches.

We propose programming patterns for Architecture-Level Software Optimizations (ALSO). We choose frequent pattern mining, one very important data-intensive application in the data mining domain, as a case study. We propose a systematic approach by identifying applicable tuning patterns. We show the generality and effectiveness of these optimization strategies by applying them to state-of-the-art implementations. We also study the sensitivity of these optimizations to inputs. Evaluation results show that on a set of datasets, the optimizations yield speedups of up to 2.1; our machine learning technique is effective at selecting the best group of optimizations.

In the architectural aspect, we propose a Near-Memory Processor (NMP), a heterogeneous architecture that couples on one chip a commodity microprocessor together with a coprocessor that is designed to run well applications that have poor locality or that require bit manipulations. The coprocessor has a blocked-multithreaded narrow in-order core, and supports vector, streaming, and bit-manipulation computation. It has no caches but has exposed, explicitly addressed fast storage. A common set of primitives supports the use of this storage both for stream buffers and for vector registers. We simulated this coprocessor using a set of 10 benchmarks and kernels that are representative of the applications we expect it to be used for. These codes run much faster, with speedups of up to 18 over a commodity microprocessor, and with a geometric mean of 5.8.
To my wife Yang.

To my parents, sister and brother-in-law.
Acknowledgments

This thesis would not have been possible without the support of many people. Many thanks to my advisor, Prof. Marc Snir. I could not have imagined having a better mentor for my Ph.D. study, and without his intelligence and perceptiveness, the knowledge and confidence that I have gained during these years would never have been so immense. Also thanks to my committee members, Jiawei Han, Josep Torrellas, and Craig Zilles, who offered guidance and support. Many thanks to my officemates and friends, Changhao Jiang and Jing Yu, for all those fruitful discussions that we had.

Finally, I would like to say “thank you” to all my family and friends, wherever they are, particularly my Mom and Dad; and most important of all, to my wife Yang, my sister and brother-in-law, for enduring this long process with me, always offering support and love.

On a different note, this work is supported by DARPA contract NBCHC-02-0056 and NBCH30390004, as part of the PERCS project.
Table of Contents

List of Tables ............................................................ viii
List of Figures ........................................................... ix
List of Abbreviations ..................................................... xi
List of Symbols ........................................................... xii

Chapter 1 Introduction and background .............................. 1
  1.1 Architecture-level software optimizations ......................... 2
  1.2 Near-memory processor ............................................. 4

Chapter 2 Programming patterns for architecture-level software optimizations ............................................. 6
  2.1 Introduction and motivation ......................................... 6
  2.2 Frequent pattern mining ............................................. 8
    2.2.1 Problem statement ............................................ 8
    2.2.2 The depth-first algorithm ................................ 8
    2.2.3 Optimization potentials .................................... 9
  2.3 Basic ALSO techniques ........................................... 11
    2.3.1 Data layout – improving the spatial locality ................ 11
    2.3.2 Tiling – improving the temporal locality .................. 12
    2.3.3 Prefetch – hiding the memory latencies ..................... 14
    2.3.4 SIMDization – improving the computation .................. 15
  2.4 ALSO patterns for frequent pattern mining ....................... 16
    2.4.1 Common optimization opportunities ........................ 16
    2.4.2 Database layout ............................................. 17
    2.4.3 Data structures ............................................. 19
    2.4.4 Data access .................................................. 22
    2.4.5 Instruction parallelism ...................................... 24
    2.4.6 Summary of ALSO patterns ................................ 24
  2.5 Case studies: LCM, Eclat and FP-Growth ......................... 25
    2.5.1 Algorithms revisited ....................................... 26
    2.5.2 Qualitative analysis on algorithm performance ............ 32
    2.5.3 The general software optimization process ................. 32
    2.5.4 LCM ......................................................... 33
    2.5.5 Eclat ....................................................... 34
    2.5.6 FP-Growth .................................................. 35
    2.5.7 Implementation details ..................................... 36
    2.5.8 Optimization results ....................................... 38
  2.6 Selecting the best group of optimizations ......................... 44
    2.6.1 Effectiveness of individual optimization on inputs ......... 44
    2.6.2 Selecting the optimal set of optimizations ................. 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.3 The support vector machine (SVM)</td>
<td>45</td>
</tr>
<tr>
<td>2.6.4 The algorithm prediction framework</td>
<td>47</td>
</tr>
<tr>
<td>2.6.5 Feature selection</td>
<td>47</td>
</tr>
<tr>
<td>2.6.6 Case study: selecting the best group of optimizations for LCM</td>
<td>49</td>
</tr>
<tr>
<td>2.6.7 Experimental results</td>
<td>50</td>
</tr>
<tr>
<td>2.7 Related work</td>
<td>54</td>
</tr>
<tr>
<td>Chapter 3 The near-memory processor</td>
<td>56</td>
</tr>
<tr>
<td>3.1 Background and motivation</td>
<td>56</td>
</tr>
<tr>
<td>3.2 Important concepts</td>
<td>58</td>
</tr>
<tr>
<td>3.2.1 Vector architecture</td>
<td>58</td>
</tr>
<tr>
<td>3.2.2 Streaming processors</td>
<td>62</td>
</tr>
<tr>
<td>3.2.3 Bit permutation instructions</td>
<td>64</td>
</tr>
<tr>
<td>3.3 Proposed architecture</td>
<td>68</td>
</tr>
<tr>
<td>3.3.1 Rationale</td>
<td>68</td>
</tr>
<tr>
<td>3.3.2 Overview of the design</td>
<td>70</td>
</tr>
<tr>
<td>3.3.3 The scratchpad</td>
<td>71</td>
</tr>
<tr>
<td>3.3.4 Instruction set architecture (ISA)</td>
<td>72</td>
</tr>
<tr>
<td>3.3.5 Other issues</td>
<td>76</td>
</tr>
<tr>
<td>3.4 Programming model</td>
<td>77</td>
</tr>
<tr>
<td>3.4.1 Processor-NMP communication</td>
<td>77</td>
</tr>
<tr>
<td>3.4.2 API for the NMP</td>
<td>77</td>
</tr>
<tr>
<td>3.4.3 Thread scheduling</td>
<td>78</td>
</tr>
<tr>
<td>3.4.4 Compilation and run-time</td>
<td>78</td>
</tr>
<tr>
<td>3.5 Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>3.5.1 Evaluation methodology</td>
<td>79</td>
</tr>
<tr>
<td>3.5.2 Main results</td>
<td>83</td>
</tr>
<tr>
<td>3.6 Related work</td>
<td>86</td>
</tr>
<tr>
<td>3.6.1 Processing in memory</td>
<td>86</td>
</tr>
<tr>
<td>3.6.2 Stream architectures</td>
<td>86</td>
</tr>
<tr>
<td>3.6.3 Multithreaded vector architecture</td>
<td>87</td>
</tr>
<tr>
<td>Chapter 4 Conclusions</td>
<td>88</td>
</tr>
<tr>
<td>Appendix A Implementations of population count function in 32-bit mode</td>
<td>90</td>
</tr>
<tr>
<td>A.1 The naive way</td>
<td>90</td>
</tr>
<tr>
<td>A.2 Popcnt by table lookup</td>
<td>90</td>
</tr>
<tr>
<td>A.3 Best scalar algorithm for popcnt</td>
<td>91</td>
</tr>
<tr>
<td>A.4 SIMDized popcnt</td>
<td>92</td>
</tr>
<tr>
<td>References</td>
<td>94</td>
</tr>
<tr>
<td>Author’s Biography</td>
<td>100</td>
</tr>
</tbody>
</table>
List of Tables

2.1  A database. Each row is a transaction. The set of all items in the database is \{a, b, c, d, e, f\}. The support of the itemset \{a, c\} is 3. ................................................................. 8
2.2  Execution time breakdown for LCM, Eclat and FP-Growth ....................................................... 11
2.3  Step one of lexicographic ordering for the database shown in Table 2.1 .................................... 18
2.4  Step two of lexicographic ordering ......................................................................................... 19
2.5  ALSO patterns ...................................................................................................................... 25
2.6  Characteristics of LCM, Eclat and FP-Growth ........................................................................ 25
2.7  Optimization patterns for LCM, Eclat and FP-Growth .............................................................. 26
2.8  IA-32 SSE prefetch instructions .............................................................................................. 37
2.9  Experimental platforms ........................................................................................................... 39
2.10 Data sets and support in the evaluation ..................................................................................... 40
2.11 Some commonly used SVM kernels .......................................................................................... 46
2.12 The selected features. \(T\) denotes the transactional database over itemset \(I\). \(|T|\) and \(|I|\) denotes the number of transactions and the number of different items respectively. ................................................................. 48
2.13 The three codes’ favorite area in the feature space. ............................................................... 50
2.14 Execution time for three example data points. “–” marks the code that does not terminate within the maximum allowed time (350 seconds). ................................................................. 52
2.15 Feature values for the three data points. ................................................................................... 53

3.1  Scalar and vector code example .............................................................................................. 59
3.2  Comparison of scalar code and vector code example in Table 3.1 .......................................... 59
3.3  Pseudocode for the convolution stage of stereo depth extraction. ............................................ 63
3.4  Bit manipulation instructions ................................................................................................... 73
3.5  Parameters of the NMP. .......................................................................................................... 80
3.6  Parameters of the memory hierarchy. ...................................................................................... 80
3.7  Parameters of the main processor. ............................................................................................ 81
3.8  Applications evaluated. ........................................................................................................... 81
## List of Figures

1.1 Scaling behavior of FP-Growth with increasing CPU frequency [GBP+05] .......................... 3
2.1 Performance for various data mining kernels ........................................................................ 7
2.2 The traversal space of itemsets ......................................................................................... 9
2.3 The candidate set \(C\) and the tested set \(T\) ........................................................................ 9
2.4 CPI for the most time consuming functions ........................................................................ 11
2.5 A snapshot of the matrix multiplication \(x = y \ast z\) before tiling (when \(i = 1\)). The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses and dark means newer accesses. .......................................................... 13
2.6 The age of accesses to the arrays \(X, Y, Z\). Note in contrast to Figure 2.5 the smaller number of elements accesses. .................................................................................................................. 13
2.7 Prefetch scheduling for a linked data structure. ................................................................. 15
2.8 Adding arrays in a scalar processor .................................................................................... 16
2.9 Adding arrays with a SIMD engine ...................................................................................... 16
2.10 Database representations .................................................................................................... 20
2.11 Aggregation for a linked list .............................................................................................. 21
2.12 Aggregation for tree ........................................................................................................... 22
2.13 Wave-front prefetch ........................................................................................................... 23
2.14 Array representation in LCM for the database in Figure 2.16 (a) ....................................... 27
2.15 Dense and sparse vertical representations for the database in Figure 2.16 (a) ................. 29
2.16 An FP-tree / prefix tree .................................................................................................... 30
2.17 A general process for software tuning .............................................................................. 33
2.18 Main data structure used in CALC\_FREQ ......................................................................... 34
2.19 Speedup of LCM, Eclat and FP-Growth on M1 and M2 ...................................................... 43
2.20 The support vector machine .............................................................................................. 46
2.21 The components and the work flow of our SVM based code selection system ............... 47
2.22 Number of times that each code version is the fastest. ...................................................... 52
2.23 Average execution time for the optimal selection, our predicted codes and the three versions of codes. ........................................................................................................................................... 54
3.1 Scalar instructions vs vector instructions. ........................................................................... 58
3.2 A generic vector architecture ............................................................................................. 60
3.3 Structure of a vector unit containing four lanes. ................................................................. 61
3.4 Imagine architecture block diagram. ................................................................................... 64
3.5 Stereo depth extraction, a stream processing example. ....................................................... 65
3.6 Kernel code structure for line 3 through 6 in main function in Table 3.3. ......................... 66
3.7 Diagram of flow of bits for PPERM 1, R1, R2, R3. \(R_2 = 0x020E160820252C33\). The numbers 2, 14, 22, 8, 32, 37, 44, and 51 are the bit positions in \(R_1\). .................................................................................................................. 67
3.8 Bit matrix multiply. .............................................................................................................. 67
3.9 GRP instruction executed with 8-bit registers. ..................................................................... 68
3.10 NMPs in a system like the IBM Power 5. ........................................................................... 69
3.11 Overall organization of the NMP. ...................................................................................... 70
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALSO</td>
<td>Architecture-Level Software Optimization</td>
</tr>
<tr>
<td>NMP</td>
<td>Near-Memory Processor</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-Level Parallelism</td>
</tr>
<tr>
<td>DLP</td>
<td>Data-Level Parallelism</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread-Level Parallelism</td>
</tr>
</tbody>
</table>
List of Symbols

\( \xi \)  Support threshold for frequent pattern mining.
\( T \)  Transactional database.
\( \mathcal{I} \)  Set of all items over \( T \).
\( \mathcal{T} \)  Tested set, the set of items that have already been tested for frequentness.
\( \mathcal{P} \)  Current solution, the set of items that are found frequent. \( \mathcal{P} \subseteq \mathcal{T} \).
Chapter 1

Introduction and background

Data-intensive applications are applications that process a large amount of data. These applications, including several key ones from the defense domain, are not running efficiently on current commodity processors.

Some applications are written in a way that do not take advantages of the underlying architecture features. Several reasons are behind that. Naive implementations make poor use of the architecture resources, in particular, the caches; software that is optimized for one architecture does not necessarily run well on another; legacy codes do not use hardware features that are available on modern architectures.

Some applications exhibit inborn access patterns that, rather than reusing the data, stream over large data structures. As a result, they make poor use of the system resources and place high demands on the main memory system. Examples in this category include vector and streaming applications. These applications have high data-level parallelism (DLP). Vector applications include typical scientific workloads, which consist of reading in large data sets, transforming them, and writing them back out to memory. There is little temporal locality in these applications. Streaming computation appears in multimedia and signal processing applications. The streaming applications are based on defining a series of compute-intensive operations (kernel functions) to be applied for each element in streams. Similarly, streaming applications run poorly on commodity processors, partially due to limited space in the processor to store temporary values. Lack of efficient architectural support, these applications are memory bound and not running efficiently on commodity processors.

In addition, some applications often perform sophisticated bit manipulation operations. For example, bit permutations are used in cryptographic applications [Sch95]. Since commodity processors do not have direct support for these operations, they are performed in software through libraries, which are typically slow.

Our software approach is to investigate causes of poor performance, and propose general approaches for architecture-level software optimizations (called ALSO patterns). We detail the study in Chapter 2. At the end of Chapter 2, we also discuss how to select the right group of optimizations that yields the best performance. For those applications that desire non-traditional architecture supports, we design a novel architecture, called Near-Memory Processor (NMP), for non-cache-friendly tasks (See Chapter 3.).
1.1 Architecture-level software optimizations

The past decade has witnessed enormous advances in processor fabrication technology and design methodologies. For a long period of time, processor speeds continued to grow at a rate up to 55% a year, whereas the memory speeds only grew at 7% a year. Computer systems suffer from the memory wall problem, where the performance of applications is increasingly determined by the memory latency.

Recently, our ability to use more gates in order to improve the performance of a single thread seems to have reached its limits, and microprocessor vendors are moving to multicore chips. Memory delays, however, tend to be higher in share-memory multiprocessors due to added contention for shared resources such as shared bus and memory modules in such systems. Memory delays are even more pronounced in distributed-memory multiprocessors where memory requests may need to be satisfied across an interconnection network.

As a result from the memory wall problem, the application performance may not scale with the processor speed. Advances in processor architectural designs do not necessarily translate to improved application performance. This is especially true for data-intensive applications, which generally access a large amount of data with poor locality.

Frequent pattern mining is a very important data-intensive application. Ghoting et al. have pointed out that even efficient frequent pattern mining algorithms are grossly under-utilizing a modern processor [GBP+05]. Figure 1.1 [GBP+05] shows the performance evolution for one very famous frequent pattern mining algorithm, FP-Growth. When the CPU frequency is scaled from 1300MHz to 3100MHz, the performance shows a sub-linear speedup. While the CPU frequency increases by a factor of 2.38, the speedup for FP-Growth saturates at 1.6, even though cache hit rates are held constant. Memory stalls are the performance bottleneck.

The Architecture-Level Software Optimization (ALSO) is our software approach to improving performance for data-intensive applications. The ALSO optimizes software performance according to the underlying architecture features of the machine on which the code is executed. These features could be those that are generally available to commodity processors or could be those that are specific to a particular architecture. By ALSO we mean optimizations that are beyond the capabilities of current compilers, because they require high level transformations that are often application specific, and often require information that is not available to compilers.

The ALSO exploits architecture potentials which translate hardware or architecture improvements to application performance. It is complementary to algorithm-level improvement and is especially important when algorithms have limited space to improve. On the other hand, ALSO has impact on the algorithm design. An example is that the outperforming of FP-Growth [HPY00] algorithm is largely due to its better data access locality and more compact data representations over traditional breadth-first algorithms.

We choose frequent pattern mining to study our software approach [WJS07]. We provide the first systematic study of architecture-level software optimizations for frequent pattern mining. We identify the performance problems due
to poor resource utilization in several highly-optimized frequent pattern mining kernels and propose programming patterns that are generally applicable. They include alternative database layout patterns that improve the spatial locality for the in-memory transactional databases; data structuring patterns for cache-conscious and optimization-friendly data structure design; and data accessing and processing patterns that improve the temporal locality, hide the memory access latency and improve the computation. Some of these patterns can be generalized and applied in other applications and domains. Among the patterns that we have proposed, the lexicographic ordering pattern and the wave-front prefetch pattern are, to our knowledge, novel patterns that have not been previously described in the literature. The aggregation, compaction, software prefetch and SIMDization patterns are for the first time to be introduced in the frequent pattern mining domain. We demonstrate the general applicability and effectiveness of these ALSO patterns by applying them to implementations of the popular LCM, Eclat and FP-growth algorithms. These algorithms have significantly different data structures and memory access patterns. Experimental results show a significant improvement in performance over the original state-of-the-art implementations.

We also study the sensitivity of these optimizations to inputs. We use machine learning technique to select the best group of optimizations and obtain good results.
1.2 Near-memory processor

As our ability to improve the performance of a single thread seems to have reached its limits, microprocessor vendors are moving to multicore chips. While current designs are of symmetric processors, as the number of cores per chip continue to increase, it is reasonable to explore heterogeneous systems with distinct cores that are optimized for different applications. A recent example of such a design is the CELL processor [PAB+05].

The advantage of a heterogeneous design is that one need not modify most of the software, as application and system code can continue running on the commodity core; code with limited parallelism can continue running on a conventional, heavily pipelined core, while code with significant data or stream parallelism can run on the new core. Each of the cores is simpler to design: the design of the new core is not constrained by compatibility requirements and good performance can be achieved with less aggressive pipelining; the design of the commodity core is not burdened by the need to handle wide vectors or other forms of parallelism. Thus, a heterogeneous system may be preferable even if, theoretically, one could design an architecture that combines both.

Three main mechanisms have been used to handle computations with poor locality: vector processing, multithreading and streaming. We show in Chapter 3 that these three mechanisms are not interchangeable: all three are needed to achieve good performance. Therefore, we study an architecture that combines all three.

Both streaming and vector processing require a large amount of exposed fast storage – explicitly addressed stream buffers and vector registers, respectively. The two approaches however manage exposed storage differently. We develop an architecture that provides one unified mechanism to manage exposed storage that can be used both for storing vectors and for providing stream buffers.

Streaming and vector provide a model where compilers are responsible for the scheduling of arithmetic units and the management of concurrency. While vector compilation is mature, efficient compilation for streaming architectures is still a research topic; existing streaming architectures cannot handle well variability in the execution time of code kernels, due to data dependent execution paths or to variability of communication time in large systems. The problem can be alleviated by using multithreading, where computational resources are scheduled “on demand” by the hardware. We show how to combine blocked multithreading with streaming and vector processing with low hardware overhead and show that a modest amount of multithreading can be effectively to achieve high performance. The NMP also enables a simpler underlying streaming compiler.

Our coprocessor is a blocked-multithreaded, narrow in-order core with hardware support for vectors, streams, and bit manipulation. It is closely coupled with the on chip memory controller. It has no caches, but high bandwidth to main memory. For this reason, rather than for its actual physical location, we call it Near-Memory Processor (NMP) [WSTT05b]. A key feature of the NMP is the scratchpad, a large local-memory directly managed by the NMP.

The main contribution of the NMP is in detailing an architecture that integrates vector, streaming and blocked
multithreading with common mechanisms that manage exposed on-chip storage to support both vectors and stream buffers. The architecture provides dynamic scheduling of stream kernels via hardware supported fine-grain synchronization and multithreading, which eases a streaming compiler’s job. To the best of our knowledge, the design is novel. The evaluation shows that all the mechanisms that are integrated in the NMP are necessary to achieve high performance.
Chapter 2

Programming patterns for architecture-level software optimizations

One very important data intensive application in the data mining domain is frequent pattern mining. Various authors have worked on improving the efficiency of this computation, mostly focusing on algorithm-level improvements. More recent work has explored architecture specific optimizations of this computation. Our goal is to provide a systematic approach to architecture-level software optimizations by identifying applicable tuning patterns. We show the generality and effectiveness of these patterns by tuning several frequent pattern mining algorithms and showing significant performance improvements. We also study the sensitivity of these optimizations to inputs and use machine learning technique to select the best group of optimizations.

2.1 Introduction and motivation

Frequent pattern mining, also known as frequent itemset mining, aims to discover groups of items that co-occur frequently in a database. This is a fundamental data mining problem with many applications. Since the introduction of this problem by Agrawal et al. [AIS93], a large number of algorithms [AIS93, AS94, GZ01, Goe02, BCG01, SON95, ZPOL97, HPY00, LPWH02, PHL+01, ZG03] have been proposed. No one algorithm dominates. Previous research has shown that the performance of these algorithms is very dependent on input characteristics [GZ03a, JGZ04, Jia07]. Figure 2.1 shows the execution time of various algorithms for one dataset. When the support changes, algorithms show a different relative performance. We have also found that the performance is very dependent on platform specific optimizations [WJS07].

We study the issue of adapting frequent pattern mining algorithms to platform characteristics. The term Architecture-Level Software Optimizations (ALSO) is used to denote such architecture specific optimizations. By ALSO we mean optimizations that are not available in current compilers, because they require high level transformations that are often application specific, and often require information that is not available to compilers.

Programming Pattern, in software engineering terminology, is a general repeatable solution to a commonly-occurring problem in software design. A pattern is not a finished design that can be transformed directly into code; it is a description or template for how to solve a problem that can be used in many different situations. We study ALSO
tuning patterns: general tuning techniques that can solve performance issues that recur in many codes; and can be easily applied by algorithm implementors.

ALSO techniques such as cache-conscious data access, prefetch and SIMDization have been applied in scientific computing, multimedia and database, but have had few applications to pattern mining. Ghoting et al. [GBP+05] have proposed optimizations for some tree based implementations. Adaptive data structures have been used in [LPWH02, LLY+03, OPPS02, OLP+03]. These papers have studied algorithms in isolation and little work has been done to develop optimizations that generalize to multiple algorithms.

We study tuning patterns that have broad applicability. These include changes in in-memory database layout to improve the spatial locality; cache-conscious and optimization-friendly data structure design; and data accessing and processing patterns that improve temporal locality, hide memory access latency and improve computation. Some of the tuning patterns, such as lexicographic ordering and wave-front prefetch are, to our knowledge, new. The aggregation, compaction, software prefetch and SIMDization patterns are for the first time used in frequent pattern mining. We demonstrate the general applicability and effectiveness of these tuning patterns by selectively applying them to three efficient and very different pattern mining algorithms, LCM, Eclat and FP-Growth, and showing significant improvements.

We also study the sensitivity of these optimizations to inputs and use machine learning technique to select the best group of optimizations.
Table 2.1: A database. Each row is a transaction. The set of all items in the database is \{a, b, c, d, e, f\}. The support of the itemset \{a, c\} is 3.

### 2.2 Frequent pattern mining

#### 2.2.1 Problem statement

Frequent pattern mining was introduced by Agrawal et al. [AIS93] in the study of association rule mining. Let \(\mathcal{I} = \{i_1, i_2, \ldots, i_m\}\) be a set of \(m\) items, and let a database \(\mathcal{T} = \{t_1, t_2, \ldots, t_n\}\) be a set of \(n\) transactions, where each transaction \(t_i\) is a subset of \(\mathcal{I}\). Any subset of \(\mathcal{I}\) is called an itemset. The projected transactional database for an itemset \(X, \mathcal{T}(X) = \{t|t \in \mathcal{T}, X \subseteq t\}\), is the set of transactions in \(\mathcal{T}\) including \(X\). The support for an itemset \(X\), denoted as \(|\mathcal{T}(X)|\), is defined as the number of transactions in the projected transactional database \(\mathcal{T}(X)\). The task of frequent pattern mining is, given a transactional database \(\mathcal{T}\) and a support threshold \(\xi\), to output all itemsets with support greater than or equal to \(\xi\).

Table 2.1 shows an example of a transactional database. Each row of the table represents a transaction in the database, which contains a set of items. The support of the itemset \(P = \{a, c\}\) is 3, because there are exactly three transactions, specifically, transaction 0, 2, and 4, that subsume \(P\).

#### 2.2.2 The depth-first algorithm

Given a database with \(m\) different items, there are potentially \(2^m\) itemsets, which form a lattice of subsets over \(\mathcal{I}\). Figure 2.2 shows an example of itemset traversal space for a database with \(\mathcal{I} = \{a, b, c, d, e\}\). A typical depth-first algorithm, starts with the initial database, and recursively creates projected databases that consist of the transactions containing a particular itemset, see Algorithm 1.

The support of an itemset is also called the frequency of that itemset. In the depth-first algorithm, the itemset \(\mathcal{I}\) is represented by a list \(\mathbf{I}\), where items are stored in a descending frequency order. By passing through all the transactions
in the database one can count the frequency of each item appearing in the database.

In the mining process, the list $I$ is divided into two separate segments, shown in Figure 2.3. The $T$ is the set of items that we have already tested for frequentness. It subsumes the current solution $P$, which is the itemset that is found frequent in the current recursion level. The set $C$ is called the candidate set, it includes all items that are stored before $T$ in $I$, i.e., all items that have frequencies greater than items in $T$. Initially, $T$ and $P$ are empty; $C$ includes all the items in $I$. The depth-first algorithm will try to add each of the items in $C$ into $T$ to form a tentative solution. If the tentative solution is frequent (the frequency is greater than or equal to $\xi$), the item is added to $P$, and the procedure is called recursively.

In line (+) of Algorithm 1, $|T(P')|$ is computed by CALCFREQ function. Note that if $T$ is the set of transactions that subsume $P$, then $T(P \cup e) = T(e)$. The database $T(P')$ in the (+) line is computed by the PROJECT function: It selects from $T$ all the transactions that subsume $P'$ and creates the projected transactional database $T'$.

### 2.2.3 Optimization potentials

We used GNU gprof [gpr] and Intel VTune Performance Analyzers [VTu] to analyze the performance for three efficient frequent pattern mining algorithms. The LCM implementation got best implementation award at the FIMI’04

$$I = \{a, b, c, d, e, f\}$$

$$C \quad T$$

Figure 2.3: The candidate set $C$ and the tested set $T$
Algorithm 1 Depth-first algorithm

\begin{algorithm}
\textsc{Depth-first FIM} ($\mathcal{T}$: transactional database, $\mathcal{P}$: current solution, $\mathcal{C}$: candidate set)
\begin{algorithmic}
\STATE // Initially, $\mathcal{P} = \emptyset$, $\mathcal{C} = \mathcal{I}$
\IF {$\mathcal{P} \neq \emptyset$} then output $\mathcal{P}$
\Foreach {$e \in \mathcal{C}$}
\STATE $\mathcal{P}' = \mathcal{P} \cup \{e\}$
\STATE $\mathcal{C}' = \{i | i \in \mathcal{C} \text{ and } i \text{ is before } e\}$
\IF {$|\mathcal{T}(\mathcal{P}')| \geq \xi$} then \hspace{1cm} (+)
\STATE $\mathcal{T}' = \mathcal{T}(\mathcal{P}')$ \hspace{1cm} (*)
\STATE call \textsc{Depth-first FIM}($\mathcal{T}'$, $\mathcal{P}'$, $\mathcal{C}'$)
\ENDIF
\ENDFOR
\ENDIF
\end{algorithmic}
\end{algorithm}

workshop [JGZ04]; the \textit{FP-Growth} got the award at the FIMI’03 workshop [GZ03a]; the \textit{Eclat} implementation is an optimized version taken from the repository of FIMI’04. These three kernels cover most common data structures and data access patterns. Same as in the FIMI workshop, we assume that all mined data can be fit in the memory. The performance data is collected on the Pentium D system described in column M1 in Table 2.9. Each core of the Pentium D processor is able to retire 3 \textit{\mu}ops per cycle, with an optimum CPI (Cycle per Instruction) of 0.33.

Table 2.2 shows the profiling results for LCM, Eclat and FP-Growth. In LCM, 54.43\% of the execution time is spent in \textsc{calcFreq} function. It counts the frequency for each item in the projected transactional database. \textsc{rmDupTrans} takes 24.58\% of the execution time: It compresses duplicated transactions in the database. In Eclat, 98\% of the total execution time is spent in \textsc{Eclat} function: It is a recursive routine to find frequent itemsets by intersecting transaction lists of itemsets. It includes both the \textsc{calcFreq} and the \textsc{project} functions shown in Algorithm 1. For FP-Growth, the \textsc{firstScan} and \textsc{secondScan} together take 85.32\% of the total execution time. Essentially being the \textsc{calcFreq} function in Algorithm 1, \textsc{firstScan} finds the set of all viable items in the FP-tree that will be used to extend the frequent itemset at that point in the search space. \textsc{secondScan} goes through the FP-tree to build a new projected FP-tree for the next step in the recursion. It is the \textsc{project} function in Algorithm 1. The access patterns of these two functions are quite similar.

Figure 2.4 shows the CPI of the most time consuming functions in the three leading frequent pattern mining codes that we studied. As we can see from the figure, there is plenty of room for performance improvements. Our general approach is to optimize memory accesses for those codes with a high CPI and cache miss rate; and to optimize the arithmetic operations for those with a low CPI and cache miss rate. The LCM and FP-Growth algorithms are clearly
Table 2.2: Execution time breakdown for LCM, Eclat and FP-Growth

<table>
<thead>
<tr>
<th>Function</th>
<th>LCM</th>
<th>Eclat</th>
<th>FP-Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>calcFreq()</td>
<td>54.43%</td>
<td>Eclat/intersect() - 98%</td>
<td>firstScan() - 63.82%</td>
</tr>
<tr>
<td>rmDupTrans()</td>
<td>25.5%</td>
<td>other - 2%</td>
<td>secondScan() - 21.5%</td>
</tr>
<tr>
<td>other</td>
<td>20.07%</td>
<td></td>
<td>14.68%</td>
</tr>
</tbody>
</table>

memory bound, as they have a high CPI, and further studies reveal that they also have high cache miss rates. As to Eclat, it has a low CPI and is computation bound. We provide details on optimization patterns that improve the performance of these codes in Section 2.4.

2.3 Basic ALSO techniques

In this section, we introduce basic ALSO techniques that have been used in the literature. These techniques help us to understand the patterns that we propose in Section 2.4.

2.3.1 Data layout – improving the spatial locality

ALSO techniques in this and the next category are mainly about how to efficiently use the caches. In modern architectures, multi-level caches are the most common and effective way to hide memory latency. They exploiting the following two features of the memory references. The temporal locality refers to the fact that a memory location that is referenced by a program at one point in time will be referenced again sometime in the near future. The spatial locality means that the likelihood of referencing a memory location by a program is higher if a memory location near it was just referenced. Higher temporal and spatial locality means fewer cache misses. A program will have reduced memory access latency if it uses memory that is already loaded in the caches.

![Figure 2.4: CPI for the most time consuming functions](image)
Algorithm 2 Code for matrix multiplication $x = y \times z$, not tiled.

```plaintext
for (i = 0; i < N; i ++)
    for (j = 0; j < N; j ++)
        r = 0;
        for (k = 0; k < N; k ++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
```

Changing data layout can enhance spatial locality, if data that are likely to be accessed together are stored in close memory locations.

2.3.2 Tiling – improving the temporal locality

Tiling, also known as blocking, is one classic approach to improving the temporal locality. The idea of tiling is to change the order of data references, so that multiple passes to large data are replaced by repeated accesses to several small amount of data, called tiles.

We take the classic example from [HP02] to show how tiling could reduce memory loads. When we deal with multiple arrays, with some arrays accesses by rows and some by columns, we often have the problem of data reuse. Instead of operating on entire rows or columns of an array, tiled algorithms operate on submatrices or tiles. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The code example shown in Algorithm 2, which performs matrix loaded into the cache before before the data are replaced. The code example shown in Algorithm 2, which performs matrix multiplication, helps motivate the optimization:

The two inner loops in Algorithm 2 read all $N \times N$ elements of $z$, access the same elements in a row of $y$ repeatedly, and write one row of $N$ elements of $x$. Figure 2.5 gives a snapshot of the accesses to the three arrays, with a dark shade indicating a recent access, a light shade indicating an older access, and white meaning not yet accessed.

The number of capacity misses clearly depends on $N$ and the size of the cache. If it can hold all three $N \times N$ matrices, then all is well, provided there are no cache conflicts. If the cache can hold one $N \times N$ matrix and one row of $N$, then at least the $i$th row of $y$ and the array $z$ may stay in the cache. Less than that, misses may occur for both $x$ and $z$. In the worst case, there would be $2N^3 + N^2$ words read from memory for $N^3$ operations. That is, for calculation of each element of $x$, a row of $y$, a column of $z$ and an element of $x$ must be retrieved from the memory. Since the $x$
Figure 2.5: A snapshot of the matrix multiplication $x = y \ast z$ before tiling (when $i = 1$). The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses and dark means newer accesses.

has $N \times N$ elements, the total number of memory access could be as many as $(N + N + 1) \ast N^2 = 2N^3 + N^2$.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size $B \times B$ by having the two inner loops compute in steps of size $B$ rather than all of $x$ and $z$. $B$ is called the tiling factor. (Assume $x$ is initialized to zero.)

Figure 2.6 illustrates the accesses to the three arrays using tiling. Looking only at capacity misses, the total number of memory words accessed could be $2N^3/B + N^2$, which is an improvement by about a factor of $B$. Provided a tile of $x$, a tile of $z$ and a tile of $y$ can be held in the cache, the capacity misses for each tile of $x$ are $B^2 + \frac{N}{B}2B^2 = B^2 + 2N B$. Since there are totally $(\frac{N}{B})^2$ tiles of $x$ to compute, the total memory accesses could be as few as $(B^2 + 2N B) \ast (\frac{N}{B})^2 = 2N^3/B + N^2$. Thus tiling exploits a combination of spatial and temporal locality, since $y$ benefits from spatial locality and $z$ benefits from temporal locality.

Although we have been aimed at reducing cache misses, tiling can also be used to help register allocation. By taking a small tiling size such that the tile can be held in registers, we can minimize the number of loads and stores in the program.

Figure 2.6: The age of accesses to the arrays $X, Y, Z$. Note in contrast to Figure 2.5 the smaller number of elements accesses.
Algorithm 3 Matrix multiplication $x = y \times z$ after tiling.

for $(jj = 0; jj < N; jj+ = B)$
    for $(kk = 0; kk < N; kk+ = B)$
        for $(i = 0; i < N; i+ +)$
            for $(j = jj; j < min(jj + B - 1, N); j+ +)$
                $r = 0$
                for $(k = kk; k < min(kk + B - 1, N); k+ +)$
                    $r = r + y[i][k] \times z[k][j]$;
                }
            $x[i][j] = x[i][j] + r$
        }
    }

2.3.3 Prefetch – hiding the memory latencies

Software prefetching is a popular technique to tolerate long memory access latencies. Prefetch instructions are issued several cycles before the requested data are accessed. Timeliness and accuracy are very important to prefetching. The prefetch instructions need to be issued early enough; the predictions to the soon-to-be-accessed addresses need to be accurate.

Figure 2.7 shows the scheduling of software prefetch for the following code.

```c
struct Node{
    struct Node* prefetch_pointer;
    struct Node* next;
    data_t data;
}
void process_node(struct Node* node){
    if (node != NULL){
        Tp:  prefetch (node->prefetch_pointer);
        Te:  process (node->data);
            process_node(node->next);
    }
}
This code traverses a linked list, processing each node accessed. Prefetch pointers `prefetch_pointer` are inserted into the list, pointing to a node that is several links ahead in the list. Software prefetches are inserted to the `process_node` function (See line `Tp`). The code on line `Te` processes the data in the node. Figure 2.7 gives the time line of the execution. Number `i` is marked in the `i`-th node in the path. `Tw` is the time for the processor to wait for the node data to arrive from memory. The node data would include the `prefetch_pointer`, the `data` to process, and the `next` pointer. Upon arrival of the node data, a non-blocking prefetch instruction is issued to prefetch some node steps away, taking `Tp` time. Finally, the `data` in the node is processed and we proceed to the next node. `Te` denotes the time spent in this final step.

We want to minimize the `Tw`, the time that the processor spent waiting for the data. For the ideal case, `Tw = 0`, the traversal is totally computation bound. Considering the prefetch, `Tw` is equal to `L_m`, the memory latency, minus the amount of time since the prefetch instruction was issued. `Prefetch distance`, denoted as `D_p` is the number of nodes to look ahead when prefetching. The optimal `D_p` is obtained when `Tw = 0`. Under this condition, the following condition `L_m - D_p(T_p + T_e) = 0` holds. We then have `D_p = \frac{L_m}{T_p + T_e}`. A more sophisticated mathematical model of prefetch distance can be found in [int04].

### 2.3.4 SIMDization – improving the computation

SIMD refers to the single instruction, multiple data execution model. The SIMDization in a modern microprocessor is to use short vector instructions for computation with high data-level parallelism. The short vector instructions were originally introduced for compute-intensive multimedia applications. At first, these instructions targeted integer computation but later were also expanded to include single and double precision floating-point computation, which makes them useful in scientific tasks.

The main idea of short vector SIMD instructions is to have multiple functional units operating in parallel, however, restricting them to work on newly introduced vector registers only. Figure 2.8 gives an example of adding two streams

![Figure 2.7: Prefetch scheduling for a linked data structure.](image)
of numbers in a scalar processor. The add operation is performed for every pair of operands in a sequential manner. In Figure 2.9, with a SIMD functional unit and extended SIMD registers, multiple operations can be performed in parallel.

2.4 ALSO patterns for frequent pattern mining

We have identified several optimization problems that occur frequently. We document the solutions to these problems as patterns in this section. Requiring application specific knowledge to apply, these patterns are high-level optimization techniques, complementary to compiler optimizations such as loop unrolling and software pipelining. These optimizations can be roughly generalized to four categories of patterns: patterns to optimize the database layout, patterns to optimize the internal data representations, patterns to optimize data accesses, and patterns to optimize data processing.

2.4.1 Common optimization opportunities

The first optimization modifies the layout of the in-memory (projected) databases. The ordering of transactions in these databases is not significant, and transactions can be permuted. We can improve the locality of accesses to the database by choosing a suitable permutation. This can have a significant impact as database transactions are often repeatedly accessed during computation. In addition, the locality property is often partially inherited by the lower-level, projected databases.

The second optimization concerns the data structure used for the database. We focus on representations that are cache-friendly, i.e., reducing cache misses; and are optimization-friendly, e.g., inserting prefetch pointers for software prefetch.

The third category includes temporal locality improvement and some memory latency hiding techniques.
Finally, arithmetic acceleration techniques can be used for computation bound applications.

We describe the ALSO patterns in detail in the following sections; we use the symbol $P_i$ to mark the $i$-th tuning pattern.

2.4.2 Database layout

This pattern is to change the database layout, defined as the ordering of transactions, to improve the data access locality during the mining process. *This optimization is used when unordered transactions are frequently accessed in a particular order.* It moves the transactions that are often successively accessed to consecutive memory locations to improve the spatial locality, reducing both cache and TLB misses.

As the transactional database is usually large, it is stored across multiple memory pages. Accessing to the transactions often involves cache misses or even TLB misses. Cache miss penalties are the time to load data from lower-level caches or main memory. TLB miss penalties are even more significant. Such misses double the number of memory accesses, as during the handling of a TLB miss, the page table entry are loaded from the memory.

The reordering process may require additional memory, the amount of which might be large when the database is large and the transactions are long.

Among all of the databases created in memory during the mining process, the initial database is the largest and is accessed most frequently. Furthermore, the layout of the initial database is preserved to some extent in the projected databases. Therefore, we focus on improving locality in the initial in-memory database.

**P1: Lexicographic ordering**

The frequency of an item is the number of occurrences of that item in the transactional database. We lexicographically orders the transactions in the in-memory initial database by following the two step preprocessing. This preprocessing is performed before the actual mining algorithm.

In step one, we order the items in each transaction in descending frequency order, see Table 2.3. In step two, we order the transactions in lexicographic order (see Table 2.4), based on the descending frequency order of the items. The frequencies for the items in the database shown in Table 2.1 are $a : 3, b : 2, c : 4, d : 2, e : 2, f : 4$. This gives a descending frequency order of $c, f, a, b, d, e$, which is the alphabet used in the second steps.

In step one, the transactional database is scanned twice. The first scanning is to count the frequency of each item. For each occurrence of an item, the correspondent frequency counter is incremented. After the first scanning, a simple sort gives us the list of items in descending frequency order. The second scanning over the database is to order the items within each transaction in descending frequency order and remove the infrequent items, i.e., those items with a frequency smaller than $\xi$ (Such item cannot appear in a frequent itemset.).
Descending frequency order: c, f, a, b, d, e.

<table>
<thead>
<tr>
<th>tid</th>
<th>transaction</th>
<th>tid</th>
<th>transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{a, c, f}</td>
<td>0</td>
<td>{c, f, a}</td>
</tr>
<tr>
<td>1</td>
<td>{b, c, f}</td>
<td>1</td>
<td>{c, f, b}</td>
</tr>
<tr>
<td>2</td>
<td>{a, c, f}</td>
<td>2</td>
<td>{c, f, a}</td>
</tr>
<tr>
<td>3</td>
<td>{d, e}</td>
<td>3</td>
<td>{d, e}</td>
</tr>
<tr>
<td>4</td>
<td>{a, b, c, d, e, f}</td>
<td>4</td>
<td>{c, f, a, b, d, e}</td>
</tr>
</tbody>
</table>

⇒

Table 2.3: Step one of lexicographic ordering for the database shown in Table 2.1

Step one is actually already included in many existing algorithms. The purpose of this step in the existing algorithms is to improve the efficiency of the mining process. We however use the result from this step to facilitate our lexicographic ordering in step two.

Step two is to lexicographically order all the transactions in the database. A sorting routine such as quick sort can be used, where the comparisons are based on the lexicographic order. Again, the alphabet for the lexicographic ordering is items ordered in descending frequency order. The order is obtained from step one.

Lexicographic ordering can be used to improve various algorithms, we give an example of its use in an array based horizontal database setting (see Section 2.4.3), which is used in LCM. We illustrate its applications in other algorithms in the case study (Section 2.5).

As described in section 2.2.2, an operation common to frequent pattern mining algorithms is to walk through the (projected) databases and construct lower-level projected databases (CALCFREQ and PROJECT functions). All transactions that contain a particular item are accessed in this process. The lexicographic ordering moves transactions containing the same item close to each other, so that spatial locality is improved; cache and TLB misses are reduced. This reduction in cache misses will be most significant when the transactions are short, as in long transactions, most of the spatial locality is already captured by storing items in each transaction in consecutive memory locations.

Consider the example in Table 2.4. We define $D(i)$ to be the number of intervals between blocks of contiguous transactions containing item $i$. This is a measure of spatial locality for an access to all the transactions containing item $i$; the greater the $D(i)$, the poorer the spatial locality. For example, in the original database, there are three transactions containing item $a$, none adjacent; therefore $D(a) = 2$. The total number of discontinuities in the original database is $\sum_i D(i) = 5$. After reordering, the total number of discontinuities is reduced from 5 to 2.

In the lexicographic layout, all transactions on the most frequent item are contiguous; transactions on the second most frequent item have at most one discontinuity; and so on. This ordering will tend to reduce the total number of discontinuities, and especially reduce discontinuities due to frequent items, thus improving locality.
Alphabet: c, f, a, b, d, e

<table>
<thead>
<tr>
<th>tid</th>
<th>transaction</th>
<th></th>
<th>tid</th>
<th>transaction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c, f, a}</td>
<td></td>
<td>0</td>
<td>{c, f, a}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{c, f, b}</td>
<td></td>
<td>1</td>
<td>{c, f, a}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{c, f, a}</td>
<td></td>
<td>2</td>
<td>{c, f, a, b, d, e}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{d, e}</td>
<td></td>
<td>3</td>
<td>{c, f, b}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{c, f, a, b, d, e}</td>
<td></td>
<td>4</td>
<td>{d, e}</td>
<td></td>
</tr>
</tbody>
</table>

⇒

D(a) = 2
D(b) = D(c) = D(f) = 1
D(d) = D(e) = 0
∑i D(i) = 5

<table>
<thead>
<tr>
<th>tid</th>
<th>transaction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c, f, a}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{c, f, a}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{c, f, a, b, d, e}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{c, f, b}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{d, e}</td>
<td></td>
</tr>
</tbody>
</table>

D(a) = D(b) = D(c) = D(f) = 0
D(d) = D(e) = 1
∑i D(i) = 2

Table 2.4: Step two of lexicographic ordering

If a bit vector is used to represent transaction occurrences in a vertical database (Section 2.4.3) then the lexicographic ordering enables another optimization, 0-escaping (see section 2.5.5 for details). For tree-based horizontal database, we lexicographically reorder transactions before tree construction (see Section 2.5.6). This improves the temporal locality for insertion and places nodes that are adjacent in a traversal path in consecutive memory locations, thus improving the spatial locality for later traversals. For tree-based algorithms, the difference between the lexicographic ordering and the depth-first order storage [GBP+05] is that the lexicographic ordering is performed as a preprocessing step before the tree is built and it optimizes both insertion and traversal operations, whereas the depth-first ordering is a reorganization of the tree structure, only to optimize the traversal.

2.4.3 Data structures

**P2: Data structure adaptation**

The data structure used to represent in-memory transactional databases can adapt to the input characteristics.

We can think of a database with n transactions and m different items as of an n × m table A: A_{ij} = 1 if transaction i contains item j, A_{ij} = 0, otherwise. There are several choices on how to represent this table.

Feature 1: The table can be stored horizontally in transaction-major order; or vertically, in an item-major order.

Feature 2: Assume a transaction-major order (some similar choices exist for item-major order). (1) One can store each row as a bit vector, so that the table is represented as a dense n × m boolean matrix; (2) alternatively, one can use a sparse representation that stores, for each row, the indices of the non-zero entries; (3) finally, one can use a prefix tree representation where shared nodes are used to represent a common prefix of several rows [HPY00]. A transaction in the database is represented by a path from the root node to the leaf node. These three representations are illustrated
in Figure 2.10, for the database shown in Table 2.1.

There are advantages and disadvantages associated with each type of data structure. The boolean matrix representation saves memory when more than \( \frac{1}{32} \) of the entries are non-zero, assuming each position takes 32 bit in the sparse matrix representation. Operations on the boolean matrices can usually be SIMDized. The population count (Section 2.5.7) for the boolean matrix representation, however, is more complicated than that for the sparse matrix representation, as one needs to count the number of 1s in the binary representation of a row.

The representation based on prefix tree is generally more compact. Transactions sharing common prefixes are compressed. There are however some additional data structure associated with prefix trees. For each node in the tree, extra storage are required for the pointers to children, parent, sibling, and the next node with the same label. For this reason, the prefix tree only saves memory when there are substantial number of transactions sharing the same prefixes. Another disadvantage of tree representation is the poor access locality, which is common in linked data structures.

Another example of the data structure adaptation pattern is to use a compression scheme whereby fewer bytes are used to represent the common cases.
This is used to improve the traversals of linked data structures, which are common in frequent pattern mining. There are two problems with such traversal. The first is that the traversal is memory latency bound, as successive memory accesses cannot be overlapped. The second is poor spatial locality, as nodes may occupy less than a cache line and successive nodes are not necessarily stored in consecutive locations.

Performance is improved by aggregating multiple consecutive nodes on a traversal path into one supernode. The number of consecutive nodes that are aggregated, is called the aggregation factor, $\eta$. Making each supernode the size of a cache line seems to be optimal. Figure 2.11 shows an example for aggregating a simple linked list, where consecutive four nodes are aggregated to a supernode, i.e., the aggregation factor is $4$. The $\#elem$ is added for each supernode to record how many elements in a supernode are valid. It is useful when nodes are inserted or deleted over time. Valid elements are stored consecutively.

One other advantage of aggregation is to save memory. As we can see from Figure 2.11 (a), 8 next pointers are needed for 8 nodes, which takes 32 bytes in a 32-bit architecture; whereas in (b), only two next pointers (8 bytes) plus two $\#elem$s (8 bytes) are needed, with a total number of 16 bytes to store the linkage information.

Suppose the cache line size is $L$; the size of the next pointer, the size of data, and the size of $\#elem$ to be $S_p$, $S_d$ and $S_\#$, respectively. Given $S_p + 2S_d + S_\# < L$ (assuming that at least we can aggregate two nodes into a cache line), the optimal aggregation factor (aggregation factor when supernode size is equal to the size of a cache line), $\eta = \lceil \frac{L-S_p-S_\#}{S_d} \rceil$. To fit a supernode in a cache line, padding is needed if $L - S_p - S_\#$ does not divide $S_d$. For a linked list with $n$ nodes, the memory compression ratio with no padding is roughly $\frac{\eta S_d + S_p + S_\#}{\eta S_d + n S_p}$, assuming $n$ divides $\eta$ for simplicity.

When aggregation is applied to trees, the nodes that are shared by multiple paths will be replicated; this partially offsets the compression achieved by using a prefix tree representation. When we aggregate a tree we have a tradeoff between a tree and a more flattened data structure. One could use 1 as the aggregation factor, which means each level is a supper level and there is no aggregation at all. On the other side of extreme, one could have an aggregation factor that is equal to the depth of the tree. This would flatten the pointer based tree structure to an array of paths. Figure 2.12 shows the aggregation of a tree structure. We compress four consecutive tree levels into one superlevel, aggregating
each path in the superlevel into one supernode.

The aggregation is efficient only when the data structure is seldom updated, as an insertion to the middle of an aggregated linked list might be expensive.

**P4: Compaction**

Compaction copies data that are scattered in memory into consecutive memory locations, to improve spatial locality. Compaction is worthwhile if the cost of copying is amortized over a large number of subsequent accesses. A small amount of extra memory is usually required during the compaction.

**P5: Pointer prefetching**

The implementations of some ALSOs require creating additional data structures. An example is the use of prefetch pointers [RS99] to improve the traversal of linked data structures. Prefetch pointers are inserted in a preprocessing stage, pointing from each node to other nodes that are likely to be accessed in the near future. Prefetch pointers allow a better overlap of memory accesses, at the expense of extra storage and preprocessing time.

### 2.4.4 Data access

Optimizations in this category focus on reducing memory bottlenecks. Some of these optimizations try to change the way how data are accessed to improve locality. Others take advantages from architecture support to hide memory latency.

**P6: Tiling**

Tiling, also called blocking [HP02], perhaps the most famous of the cache optimizations, tries to reduce misses by improving temporal locality. It is used when large data structures are accessed repeatedly.
Tiling for dense matrix operations shown in Section 2.3.2 could be applied to variants of frequent pattern mining that use such matrices. If such implementations are memory bound, tiling could reduce the memory pressure. Tiling for trees is proposed in [GBP+05].

**P6.1: Tiling for sparse representations.** Sparse matrices are commonly used to represent the database. Temporal locality is poor when a large database is repeatedly traversed. Researchers have proposed tiling for sparse matrix vector multiplications [Im00, IYV04, IY01]. This work is, however, tied to sparse matrix and dense vector operations and does not directly apply to frequent pattern mining. Our basic idea for tiling is to slice the sparse matrices into horizontal tiles according to the row range and then to process one tile at a time, with an outer loop that walks through tiles and an inner loop that traverses entries within a tile. See section 2.5.4 for an example. The disadvantage of tiling is the overhead for the added level of loop nesting.

**P7: Software prefetching**

Prefetching, exploiting the overlap of processor computation with data accesses, is an effective approach to tolerate memory latencies. Prefetching can be either hardware-based or software-based. In software prefetching, prefetch instructions, loading data to the cache in a non-binding fashion, are inserted several cycles before their corresponding memory instructions. Software prefetching can be used for linked data structure, where hardware prefetching does not work well. Software prefetching can be performed by following the pre-inserted prefetch pointers [RS99]. Mispredicted prefetches, however, may degrade the performance.

**P7.1: Wave-front prefetching.** Arrays of short linked lists (see Figure 2.13) are common in frequent pattern mining. The common access pattern to this data structure is to traverse all nodes. Existing linked list prefetch algorithms only have good performance when the linked lists are long and do not apply to our case. Instead, we propose to use wave-front prefetching. See Algorithm 4. The basic idea is that we can prefetch entries from different linked lists in the same
Algorithm 4 Wave-front prefetch algorithm

 Traverse \((P: \text{array of linked lists})\)

 For \(i \leftarrow 0\) to \(n - 1\)

 Prefetch\(P[i + 2] \rightarrow \text{next} \rightarrow \text{next}\)

 Prefetch\(P[i + 4] \rightarrow \text{next}\)

 Prefetch\(P[i + 6]\)

 Traverse linked list \(P[i]\)

iteration. In Figure 2.13, the numbers over the arrows are the iteration numbers when the correspondent entries are prefetched; the indices on the left indicate the iteration when the linked list is traversed. Suppose the memory latency is less than the time to traverse two short linked lists, then we can prefetch three links in each iteration as shown in Figure 2.13. At the time when entries need to be prefetched, their addresses have already been loaded by previous prefetches.

2.4.5 Instruction parallelism

Optimizations in this category focus on improving instruction parallelism, for computation bound kernels.

*P8: SIMDization*

SIMD instructions are available on most of the commodity processors. SIMDization can accelerate computation bound applications. Memory prefetch instructions are also available in the SIMD instruction set. The SIMDization optimization, however, requires sufficient data-level parallelism in the algorithm and one needs to handle memory alignment problems.

2.4.6 Summary of ALSO patterns

Table 2.5 summarizes the ALSO patterns and shows what improvements these optimizations can provide.

The *lexicographic ordering* pattern can improve spatial locality, as it moves transactions that are likely to be accessed together to closer memory locations. One can apply lexicographic ordering in tree based implementations to improve temporal locality (Section 2.5.6). Such reordering in an algorithm like Eclat (Section 2.5.5), could cluster data to be computed, enabling 0-escaping. This reduces computation pressure.

The *data structure adaptation* pattern adapts data structure according to input characteristics. One may use data structure with high spatial locality, such as arrays; or may use more compact representations such as trees to save
memory.

The aggregation pattern packs linked data in consecutive memory memory locations. It improves spatial locality for traversal, and reduces unnecessary memory loads due to small node size.

By Compacting frequently accessed data into consecutive locations, we improve the spatial locality. As the data now take fewer cache lines, there is less chance for cache thrashing. The temporal locality is improved.

### 2.5 Case studies: LCM, Eclat and FP-Growth

We selected three highly optimized frequent pattern mining kernels to evaluate the applicability and effectiveness of our ALSO patterns. They cover most efficient algorithm space and data structure design choices. The LCM implementation got best implementation award at the FIMI’04 workshop [JGZ04]; the Eclat implementation is an optimized version taken from the repository of FIMI’04 [Bor04]; FP-Growth is an efficient implementation of the FP-Growth algorithm. The Eclat implementation that we studied uses a bit vector data structure for the transactional database. Table 2.6 shows the characteristics of the three kernels evaluated. We did not cover breadth-first search

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Database type</th>
<th>Data structure</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCM</td>
<td>horizontal</td>
<td>array</td>
<td>memory</td>
</tr>
<tr>
<td>Eclat</td>
<td>vertical</td>
<td>bit vector (array)</td>
<td>computation</td>
</tr>
<tr>
<td>FP-Growth</td>
<td>horizontal</td>
<td>tree</td>
<td>memory</td>
</tr>
</tbody>
</table>

Table 2.6: Characteristics of LCM, Eclat and FP-Growth
<table>
<thead>
<tr>
<th>Patterns</th>
<th>LCM</th>
<th>Eclat</th>
<th>FP-Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexicographic ordering</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Data structure adaptation</td>
<td>—</td>
<td>O</td>
<td>√</td>
</tr>
<tr>
<td>Aggregation</td>
<td>√</td>
<td>—</td>
<td>√</td>
</tr>
<tr>
<td>Compaction</td>
<td>√</td>
<td>—</td>
<td>√</td>
</tr>
<tr>
<td>Pointer prefetching</td>
<td>—</td>
<td>—</td>
<td>√</td>
</tr>
<tr>
<td>Tiling</td>
<td>√</td>
<td>—</td>
<td>O</td>
</tr>
<tr>
<td>Software prefetch</td>
<td>√</td>
<td>—</td>
<td>√</td>
</tr>
<tr>
<td>SIMDization</td>
<td>—</td>
<td>√</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2.7: Optimization patterns for LCM, Eclat and FP-Growth

algorithms, such as Apriori [AS94], because the depth-first search algorithms are generally considered to be more efficient and our study is focusing on kernels with different data representations, rather than a study on different algorithms. We applied several locality and memory optimization patterns on LCM and FP-Growth, and mainly used computation optimization patterns on Eclat. Table 2.7 shows the patterns that we have studied for these three kernels. The “√” marks those patterns that we have applied in the case studies. The “○” marks the optimizations that have already been proposed in the literature, which we did not incorporate in the evaluation. “—” are the patterns that we have not applied.

### 2.5.1 Algorithms revisited

**LCM**

LCM [UAUA03, UKA04] (*Linear time Closed itemset Miner*) algorithm creates projected databases only for frequent closed itemsets. An itemset $P$ is a *closed itemset* if it is not properly contained in an itemset $Q$ with the same support as $P$. LCM generates the remaining frequent itemsets by enumeration. This technique is called *hyper-cube decomposition*:

1. Suppose that $P$ is a frequent closed itemset, $P \cap Q = \emptyset$ and any transaction that contains $P$ also contains $Q$. Then $P \cup Q'$ is a frequent itemset, for any $Q' \subseteq Q$.
2. If $P$ is an itemset, then there is an itemset $Q \subseteq P$ so that $Q$ is a closed itemset with the same support as $P$.
3. The itemsets can be partitioned so that each component consists of all the itemsets $\{ P \cup Q' : Q' \subseteq Q \}$, where $P$ is closed, and $P \cap Q = \emptyset$.

This *hyper-cube decomposition* can significantly speedup the mining process when projected databases have many
LCM uses arrays to represent projected transactional databases. Figure 2.14 illustrates the data structure used by LCM to represent the database shown in Figure 2.16 (a). The transactional database is a list of transactions, where each transaction is represented by an array containing item IDs. The OccArray has one record for each item; the record contains an array of pointers to the transactions that include the corresponding item. In most cases, the array representation of LCM is less compact than the FP-tree in FP-Growth. However, the advantage of LCM is that it has more spatial locality on memory accesses than FP-Growth where the extensive use of pointers and associated pointer-chasing during the FP-tree traversal can degrade the performance.

Algorithm 5 gives the skeleton of the LCM algorithm. \( P \) is the current solution, the transactional database is projected so that it only contains transactions that contain \( P \). \( \text{CLOSE} \) is a set of items that have the same frequencies as the current solution in the projected database. \( C \) is the set of items whose frequencies are smaller than current solution, but greater than the support threshold \( \xi \). The frequent itemsets can be generated by enumerating current solution unioned by each set in the power set of \( \text{CLOSE} \).

Eclat

Eclat [ZPOL97] is another well-known depth-first frequent pattern mining algorithm. Eclat uses a vertical (item-major) representation of the database; each column record corresponds to an item, or an itemset, and lists the transactions containing this item (resp. itemset). During the recursive depth-first-search of the subset lattice, records are intersected to compute the record corresponding to the union of the two corresponding itemsets (see line (∗) of Algorithm 6).

Eclat can store the itemset records either in sparse or in dense format. Figure 2.15-(a) shows the dense representation of the database in Figure 2.16, where each item record is a bit vector, and 1 indicates the occurrence of an item in a transaction.
Algorithm 5 LCM algorithm

LCM ($T$: transactional database, $I$: set of all items)

for $i \leftarrow 0$ to $|I| - 1$
    call LCMTER($T$, $i$)

LCMTER ($T$: transactional database, $max$: max item, $\mathcal{P}$: current solution)

// $\xi$: threshold
// $\mathcal{P}$: current solution, $\mathcal{P} = \{\}$ initially.
// $\mathcal{CLOSE}$: closed itemset, $\mathcal{CLOSE} = \{\}$ initially.
// $\mathcal{C}$: candidate itemset, $\mathcal{C} = \{\}$ initially.
// $\mathcal{INF}$: set of infrequent items, $\mathcal{INF} = \{\}$ initially.
$\mathcal{P} \leftarrow \mathcal{P} \cup \{\max\}$ // Add current item to the tentative solution.
$\mathcal{T}(\max)$ returns all transactions in $\mathcal{T}$ that subsume $\max$.
call CALC_FREQ($\mathcal{T}$, $\max$, $\mathcal{T}(\max)$)

foreach item $i < \max$
    if ($|\mathcal{T}(i)| = |\mathcal{T}(\max)|$) $\mathcal{CLOSE} \leftarrow \mathcal{CLOSE} \cup \{i\}$
    elseif ($\xi < |\mathcal{T}(i)| < |\mathcal{T}(\max)|$) $\mathcal{C} \leftarrow \mathcal{C} \cup \{i\}$
    else $\mathcal{INF} \leftarrow \mathcal{INF} \cup \{i\}$
    output $\mathcal{P} \cup 2^{\mathcal{CLOSE}}$ as frequent
$\mathcal{TransTable} = \text{rebuild} (\mathcal{T})$ // Remove items in $\mathcal{INF} \cup \mathcal{CLOSE}$.
$\text{rmDupTrans} (\mathcal{TransTable})$ // Remove duplicated transactions.
foreach item $i$ in $\mathcal{C}$
    LCMTER ($\mathcal{TransTable}$, $i$, $\mathcal{P}$)

CALCFREQ($\mathcal{T}$: transactional database, $\max$: max item, $\mathcal{occ}$: $\mathcal{T}(\max)$)

// This function calculates $\mathcal{T}(i)$ for all $i < \max$.
foreach $t \in \mathcal{occ}$
    foreach $i \in t$ and $i < \max$
        $\mathcal{freq}[i]++$ // $\mathcal{freq}[i]$ is $|\mathcal{T}(i)|$. 
transaction. Bit vector representation allows direct use of bit operation instructions. However, when there are too few 1’s in the bit matrix, it is more efficient to represent the bit matrix in sparse format as shown in figure 2.15-(b), where each record is a list of transaction IDs. [ZG03] proposed an optimization for vertical algorithms based on $diffset$. The idea is to only keep track of the differences in the transaction IDs of a candidate itemset from its generating itemsets. The $diffset$ idea can significantly reduce the memory usage of Eclat.

**FP-Growth**

FP-Growth was first proposed by Han et al. [HPY00]. This algorithm uses an augmented prefix tree, called $FP-tree$

---

**Algorithm 6 Eclat algorithm**

ECLAT ($M$: transactional database)

For $i ← n – 1$ down to 0

For $j ← 0$ to $i – 1$

// $M_i$ is the $i$-th row of matrix $M$

$newRow ← M_i \land M_j$  

// $CALC\text{FREQ}$

$support ← \text{popcnt}(newRow)$

If $support \geq \xi$

output $I_i \cup I_j$ as frequent  

add $newRow$ to $M^r$

ECLAT($M^r$)
(Frequent Pattern Tree) to represent in a compact way the database. The FP-tree is very efficient at compressing databases when many transactions share common prefixes, as shown in Figure 2.16 (b). The correspondent database is shown in Figure 2.16 (a). The FP-Growth algorithm proceeds by performing two data scans over the original database; the first one counts the number of occurrences of each item, and the second one builds the initial FP-tree. Then, it recursively builds smaller FP-trees that represent projected databases, consisting of all transactions containing a particular itemset. Experimental results [GZ03b, GBP+05] show that FP-Growth spends most of the time building and traversing the FP-trees. To reduce this overhead, the authors of [GZ03b] implemented a variant of the original FP-Growth algorithm where a 2D array that counts the frequencies of all pairs of frequent items is constructed at the same time as each FP-tree. This optimization results in significant performance savings when the database is sparse. The implementation in [GZ03b] only uses the 2D array optimization when the database appears to be sparse. Another potential problem of the FP-Growth algorithm is that each node in the FP-tree requires four pointers, one to the parent, one to the child, one to the sibling to the right, and another to the next node with the same item. The extra pointers are for representing a general tree by using a binary tree. These pointers may add significant overhead to the traversal of the FP-tree and increase the memory consumption.

Algorithm 7 summarizes FP-Growth [GBP+05] algorithm.

<table>
<thead>
<tr>
<th>TID</th>
<th>(Ordered) frequent items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a c f m p</td>
</tr>
<tr>
<td>2</td>
<td>a c f b m</td>
</tr>
<tr>
<td>3</td>
<td>a c f m p</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>f b</td>
</tr>
<tr>
<td>6</td>
<td>c b p</td>
</tr>
</tbody>
</table>

Figure 2.16: An FP-tree / prefix tree

Algorithm 7
Algorithm 7 FP-Growth algorithm

FP-GROWTH ($T$: FPTree, suffix: itemset)

If tree has only one path

Output $2^\text{path} \cup \text{suffix}$

Else

Foreach frequent one item $e$ in the header table

Output the $\{e\} \cup \text{suffix}$ as frequent

FIRSTSCAN: Use the header list for $e$ to find all frequent items in conditional pattern base $C$ for $e$

SECONDSCAN: If we find at least one frequent item in the conditional pattern base, use the header list for $e$, and $T$ to generate conditional prefix tree $N$

If $N \neq \{\}$ then

FP-GROWTH($N$, $\{e\} \cup \text{suffix}$)
2.5.2 Qualitative analysis on algorithm performance

In order to achieve the best performance, one would want to select the fastest algorithm. As we have mentioned, the performance of frequent pattern mining algorithms are input dependent. We however have some qualitative understanding of input features that may cause one algorithm to run faster than another [Jia07]. Since all these algorithms traverse the search space in the same order (depth-first order), the major difference between them is in the data structure used to represent the database. FP-Growth uses FP-trees, LCM uses arrays, and Eclat uses bit matrices. The bit matrix representation is more efficient when the database is large and dense. However, since the Eclat implementation we use does not implement the diffset idea, the bit matrix gets sparser when recursing down the search space, and becomes more and more inefficient compared with LCM’s arrays and FP-Growth’s FP-tree. Hence, intuitively, Eclat will have better performance than LCM and FP-Growth when the input database is large, dense, and the search space is shallow.

The major difference between LCM and FP-Growth is the data structure and the number of projected databases. LCM only recurses for closed itemsets and enumerates all other frequent itemsets by using the hyper-cube decomposition. If the number of frequent closed itemsets is much smaller than the number of frequent itemsets, LCM should have better performance. When the number of frequent closed itemsets is close to the number of frequent itemsets, the representation of the database (arrays versus FP-tree) is the main factor determining the performance difference between the two algorithms. If the FP-tree representation can effectively compact the database so that the compressed tree can fit in the cache, while the array representation exceeds it, FP-Growth is likely to perform better. The problem of FP-Growth is that it uses several pointers for each node in the tree, and if the compression ratio of the FP-tree structure is not big enough, the FP-tree may end up using more memory than the array. In addition, traversing the trees in FP-Growth requires extensive pointer chasing, which results in less spatial locality and more non-overlapped memory accesses than the array structure of LCM.

2.5.3 The general software optimization process

Figure 2.17 shows a general process for software tuning. The software tuning is an iterative process. After the program to be tuned is selected, the first step is to find the hotspots, which are the areas of the application that have intense activity (execution time). According to Amdahl’s law, hotspots are the places to start the optimization. Profilers are used to identify procedures that take most fraction of the execution time.

The next step is to investigate causes of the hotspots. The reasons could be inefficient arithmetic operations, branches mispredictions, cache misses, etc. The VTune performance analyzers [VTu], can be used to collect program characteristics such as cache misses or branch mispredictions.

The hotspots are modified to improve the performance. For programs with high cache miss rates, locality improve-
ment can be used. For computation bound application, methods such as SIMDization can be used.

Modified program is then re-run to evaluate the performance. If the performance is not satisfying, one needs to repeat the above process.

We follow this optimization process, using GNU gprof for the profiling and VTune for program characterization. We found both LCM and FP-Growth are memory bound; Eclat is computation bound. Next we studied the implementations in detail and applied several optimization patterns.

### 2.5.4 LCM

Since LCM (Linear time Closed itemset Miner) [UKA04] is memory bound, we focus on patterns that improve the memory performance.

Figure 2.18 shows the main data structure that is traversed by the CALCFREQ function. This function takes 54.43\% of the total execution time. The data structure consists of a transaction-major sparse array that represents the database, augmented by an item-major sparse array OccArray that is used for speeding up the construction of projected databases. Each column (called occ, shown as shaded column) stores pointers to the headers of transactions. These transactions contain the corresponding itemset. For each call of CALCFREQ, the execution traverses one of these columns (an occ), follows the pointers to transaction headers and accesses all the items in these transactions. Essentially, all transactions containing one itemset are accessed. Pointers dereferenced in this process are shown as dashed arrows in Figure 2.18.

We use lexicographic ordering to improve the spatial locality of the initial database. Transactions are reordered so that accesses to the transactions subsuming one itemset (in CALCFREQ) are liked to be stored in consecutive memory locations.

The frequency counters that are frequently accessed in CALCFREQ are not in contiguous locations. They are structured with the OccArray. By compaction, the frequency counters are moved to contiguous memory locations,
thus improving the locality and reducing the cache and TLB misses.

As CALCFREQ has extensive the traversals of a list of short linked lists, we use wave-front prefetch for pointers in occ array and pointers in transaction headers.

The function CACLREQ is called from a loop that invokes CALCFREQ for each occ (one column of OccArray). For each run of CALCFREQ, in the worst case, the whole database is scanned, with little cache reuse. Tiling for sparse representations could be done across invocations of CALCFREQ in the following way: The array OccArray is split into tiles (separated by dark lines in Figure 2.18). Each tile contains the transactions within a particular offset range. The inner loop performs all the CALCFREQ computations for one tile; the outer loop iterates over all tiles. We choose the tile size to fit in the L1 cache.

Another function that takes 25.5% of the total execution time is RMDUPTRANS. It compresses identical transactions in the database. In the original implementation, bucket (radix) sort is used to find these transactions. A linked list is used to link all the transactions that fall into one bucket. As the linked list is mostly read only, we use aggregation to reduce dereferences and improve spatial locality.

### 2.5.5 Eclat

The Eclat algorithm [Bor04] uses a vertical, dense bit matrix representation. The columns represent initially the occurrences of items in transactions; as the algorithm proceeds, the columns represent the occurrences of itemsets in transactions. The and of the bit vectors for two itemsets computes the bit vector for the union of the two itemsets. 98%
of the total execution time is spent in these vector ands and in counting the number of ones in the resulting vectors (frequency count).

By lexicographic ordering the initial transactions, the 1s in the bit vectors for the most frequent items are clustered. In particular, the 1s for the most frequent item are consecutively stored at the beginning of the vector. The lexicographic ordering enables the 0-escaping. The idea of 0-escaping is to skip intersecting and frequency counting on the bit vector ranges where either operand vectors are all 0s. This is achieved by storing, for each vector, the start and end position of a 1-range, which includes all the 1s in the bit vector. The ranges are initialized by identifying the first and last 1 in each item bit-vector and updated by intersecting the corresponding 1-ranges when two bit vectors are anded. Then the intersection and frequency counting are performed only within the computed 1-ranges, skipping 0s at the beginning and end of the intersecting vectors. The reordering improves the performance of 0-escaping, as the 1s are moved together and the 1-range for the correspondent bit vectors becomes shorter; fewer operations need to be performed. Note that the 1-ranges thus computed are conservative, but not necessarily optimal.

There is plenty of data-level parallelism in Eclat. Clearly, the bit vector intersection can be SIMDized. In the original implementation, table lookups are used to count the number of 1s (population count) in the bit vector. The table lookup is an indirect load, which cannot be SIMDized. We use computations to count the frequency of 1s, which can be easily SIMDized (Section 2.5.7).

2.5.6 FP-Growth

FP-Growth [HPY00] uses an augmented prefix tree known as the FP-tree (see Figure 2.16) to represent the database. The most common access pattern is to follow pointers in head of node links to access the nodes labeled by the same item (shown as dashed arrows in Figure 2.16). For each node accessed, the path from that node to the root is then traversed.

The FP-Growth algorithm has a high CPI and cache miss rate; it is memory bound. Several optimizations have been proposed in [GBP+05], which include initial database reorganization, tiling, etc. We propose to use lexicographic ordering, data structure adaptation, aggregation and software prefetch to improve the performance. These new techniques are complementary to the optimizations that have been previously studied.

A lexicographically ordering of the transactions for FP-Growth provides two benefits: First, the tree construction is more cache efficient. The tree building process inserts transactions one by one. After the reordering, as each transaction shares many items with the previous one, most of the nodes accessed during an insertion are already in the cache. Second, pairs of parent node and child node, which are often accessed together during traversals, are likely to be stored next to each other. It is because after the reordering, the insertions would make the tree expands in a depth-first manner. Recall one of the common access patterns to the tree is to go up the tree from a intermediate node
to the root, storing parent and child node into consecutive memory locations would improve the spatial locality.

A useful data structure adaptation is to represent the item ID of a node with fewer bytes, using differential encoding: one stores the difference between the local item ID and the ID of the parent node; this can usually be stored in a single byte, with an escape code to handle the exception cases. This reduces the node size and memory requirements dramatically.

Aggregation can be used in FP-Growth to improve the spatial locality of tree traversals. As nodes that are shared between paths need to be copied, the aggregated tree requires more memory. We find that, when combined with data structure adaptation aforementioned, the memory requirements are moderate.

Prefetch pointers can also be inserted to help software prefetch. For each intermediate and leaf node, we insert a pointer to its ancestor that is steps away in the upper level. During the traversal, the ancestors are prefetched before they are accessed in the future.

2.5.7 Implementation details

SIMDization of population count

The population count refers to the function that returns the number of 1s in a word’s binary representation. This function is extensively used in LCM. We SIMDed the population count in our LCM implementation. The following BitCount32 function [bit] is the scalar version of the code. As each 32-bit value can be calculated independently, this function can be SIMDed. The SIMDized version of BitCount32 can be found in Appendix A.4.

```c
unsigned BitCount32(unsigned b)
{
1:  b = (b & 0x55555555) + (b >> 1 & 0x55555555);
2:  b = (b & 0x33333333) + (b >> 2 & 0x33333333);
3:  b = (b + (b >> 4)) & 0x0F0F0F0F;
4:  b = b + (b >> 8);
5:  b = (b + (b >> 16)) & 0x0000003F;
    return b;
}
```

Line 1 of the above code partitions the integer into groups of two bits, computes the population count for each 2-bit group and stores the result in the 2-bit group. b & 0x55555555 masks out all the odd bits. b >> 1 & 0x55555555 does the same thing for all the even bits. After line 1, each 2-bit group in b stores the number of 1s for that two bits. Line 2, 3, 4, 5 are performing the same procedure for each 4-bit, 8-bit, 16-bit and 32-bit respectively.
### Instruction Description

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefetcht0</td>
<td>Temporal data; prefetch data into all cache levels.</td>
</tr>
<tr>
<td>prefetcht1</td>
<td>Temporal with respect to first level cache; prefetch data in all cache levels except 0th cache level.</td>
</tr>
<tr>
<td>prefetcht2</td>
<td>Temporal with respect to second level cache; prefetch data in all cache levels, except 0th and 1st cache levels.</td>
</tr>
<tr>
<td>prefetchnta</td>
<td>Non-temporal with respect to all cache levels; prefetch data into non-temporal cache structure, with minimal cache pollution.</td>
</tr>
</tbody>
</table>

Table 2.8: IA-32 SSE prefetch instructions

In the SIMDization process, bit vectors need to be aligned to 128-bit addresses. Each 128 bits are grouped to one block and processed by SSE2 instructions. At the end of bit counting, sums are accumulated in one 128-bit MMX register, as four 32-bit integers. The final step is to add these four integers to get the population count for the whole bit vector.

**Prefetch instructions for modern microprocessors**

This section includes descriptions of prefetch instructions in IA-32 SSE and 3DNow, which can be found in [tea].

**[IA-32 SSE]** The IA-32 Streaming SIMD Extension (SSE) instructions are used on several platforms, including the Pentium III, Pentium 4 [ia3], and IA-32 support on IA-64 [ita]. The SSE prefetch instructions are included in the AMD extensions to 3DNow! and MMX used for x86-64 [AMD00b].

The variants of SSE prefetch instructions are shown in Table 2.8.

There are no alignment requirements for the address. The size of the line prefetched is implementation dependent, but a minimum of 32 bytes.

**[3DNow!]** The 3DNow! technology from AMD extends the x86 instruction set, primarily to support floating point computations. Processors that support this technology include Athlon, K6-2, and K6-III.

The instructions PREFETCH and PREFETCHW prefetch a processor cache line into the L1 data cache[AMD00a]. The first prepares for a read of the data, and the second prepares for a write.

There are no alignment restrictions on the address. The size of the fetched line is implementation dependent, but at least 32 bytes.

The Athlon processor supports PREFETCHW, but the K6-2 and K6-III processors treat it the same as PREFETCH. Future AMD K86 processors might extend the PREFETCH instruction format.
for (i = 0; i < n; i ++)
{
    a[i] = a[i] + b[i];
    _builtin_prefetch (&a[i + j], 1, 1);
    _builtin_prefetch (&b[i + j], 0, 1);
}
the Text Research Collection [ap94]. Table 2.10 shows the data sets and the support that we use in the evaluation. We choose WebDocs and AP, because other available real world data sets are too small.

The baselines of our speedup are the best implementation of FIMI’04: LCM, an optimized version of Eclat from FIMI’04 and an efficient implementation of FP-Growth. The baseline running times are listed in Figure 2.19. The speedup is based on overall execution time.

Figure 2.19 shows the speedup of the optimized LCM, Eclat and FP-Growth on systems M1 and M2. In these figures, Lex means the speedup we get after we lexicographically reorder the initial database; Reorg refers to the data structure optimizations such as aggregation and compaction; Pref refers to software prefetching; Tile and SIMD are the tiling and SIMDization pattern respectively. We first apply each applicable ALSO pattern to each algorithm to see the benefit of a single pattern. Then we test the performance for the code that incorporates all applicable patterns. For each cluster of columns, the second column from the right, labeled all, is the performance after we apply all applicable patterns; the rightmost column, labeled best, is the best performance that we can get by selectively applying the patterns. For most of the cases best and all are the same, indicating that each of the optimizations provides some benefit, when combined with all others. In some cases, for example in Figure 2.19(a) data set DS4, the best optimization is not all. Instead, it is the combination of prefetch and data structure patterns. The texts above the best bars show the combination of patterns that yields the best performance.

We can immediately see that there is no single best algorithm. For the baselines, Eclat performs the best on DS3, while for other data sets, LCM is the fastest algorithm. The FP-Growth also has a competitive performance, and in some cases is close to optimal.

We see an overall performance improvement for the best combination of patterns, ranging from 1.08 to 2.1. We also see a significant performance improvement for the application of each individual pattern. To be specific, the lexicographic ordering provides up to 1.5 speedup. Software prefetch gives up to 1.3 speedup. The SIMDization
provides a speedup between 1.25 and 1.45 on M1. In FP-Growth, data structuring techniques, particularly, data structure adaptation and tree aggregation give a speedup of 1.6. Tiling in LCM gives a speedup of up to 1.75. The tiling for FP-Growth has been studied elsewhere [GBP+05], and yields a speedup of about 2.

The effectiveness of optimizations is input dependent. For the inputs shown in Figure 2.19(a), tiling in most cases provides the most significant speedup to LCM, in particular, for DS1 and DS2, tiling produces a speedup of over 1.5. In DS4, tiling, however, produces almost no speedup. Each software optimization have some associated cost, which can negate its benefit. DS4 is a very sparse data set, where transactions containing one item are scattered over memory. In this sparse data set, tiling does not introduce much data reuse. The lexicographic ordering is not performing well in FP-Growth for DS4, because the data set contains so many transactions that lexicographic ordering is very time consuming.

In general, software prefetch and aggregation work better for long linked data structure, as there is more potential for latency reduction. For example, in FP-Growth, a greater average transaction length would be an indication of deeper FP-tree, i.e., longer linked structure. Lexicographic ordering would work better if the order of transactions in the original input database is random. One could define a metric that capture the clustering of the input transactions. Tiling would work better when the transactions are clustered, as it tends to have more cache reuse in this case.

The optimization results are also platform dependent. Figure 2.19(b) shows the same experiments as in Figure 2.19(a) but on a different platform M2. Although optimizations have similar impact on the performance, the magnitude is different. In particular, in Figure 2.19(c) and Figure 2.19(d), the SIMD performance of M2 is not so significant as that of M1, providing less than 1.2 speedup for the best case.

Finally, the optimizations seem to be dependent. Several optimizations may have the same objective (e.g., improving spatial locality). If one optimization is sufficiently effective, then other optimizations may add little value, while still incurring overhead.

Our results show that for Eclat and FP-Growth, there is on each platform one code that is best for all inputs, while LCM requires different codes for different inputs. However, due to the small number of experiments one cannot attach too much significance to this conclusion.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>DS1</th>
<th>DS2</th>
<th>DS3</th>
<th>DS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>T60I10D300K</td>
<td>T70I10D300K</td>
<td>WebDocs</td>
<td>AP</td>
</tr>
<tr>
<td># transactions</td>
<td>300K</td>
<td>300K</td>
<td>500K</td>
<td>1.8M</td>
</tr>
<tr>
<td>Support used</td>
<td>3000</td>
<td>3000</td>
<td>50000</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 2.10: Data sets and support in the evaluation
We studied in Section 2.6 how these optimizations are sensitive to input features, and how to select the right group of optimization techniques.

(a) LCM on M1, baseline in seconds (77, 169, 90, 36)

(b) LCM on M2, baseline (74, 159, 93, 35)
(c) Eclat on M1, baseline (137, 270, 50, 751)

(d) Eclat on M2, baseline (142, 285, 50, 887)
Figure 2.19: Speedup of LCM, Eclat and FP-Growth on M1 and M2
2.6 Selecting the best group of optimizations

As shown in Section 2.5.8, the effectiveness of a single optimization is input dependent; further more, the optimal set of optimizations is input dependent. Given a running environment, the characteristics of the input database, and the support threshold decide not only the overall performance of the implementation, but also the amount of speedups that the architecture-level software optimizations could offer. All these propose a problem of selecting the right set of optimizations according to inputs.

In this section, we illustrate how to choose the best group of optimizations according to the inputs. In Section 2.6.1, we study the sensitivity of a single optimization to inputs. In Section 2.6.2, Section 2.6.3, Section 2.6.4 and Section 2.6.5, we introduce our idea of using machine learning to select the right set of optimizations. We use LCM as a case study and present our prediction results in Section 2.6.6 and Section 2.6.7.

2.6.1 Effectiveness of individual optimization on inputs

Although the performance of a group of optimizations might show different behavior, studying each individual optimization helps us to understand the optimization selection problem.

The lexicographic ordering optimization improves the access locality to those transactions subsuming the most frequent items. When applied to algorithms, one should expect better speedups for databases with highly frequent items. The frequency of the most frequent items are important.

Data structure adaptation is generally effective. It adapts data structure according to inputs. It would work less efficiently if the input characteristics are close to the cross point of adaptation. The cross point is the place where transition from one data structure to another takes place.

Aggregation and software prefetch are optimizations to linked data structures. They are efficient if the linked list is long and the data structure is seldom updated. In terms of frequent pattern mining, long transaction, or large number of transactions usually means long linked data structures.

Tiling is generally effective. It improves the temporal locality for data that are repeatedly traversed. The more frequent the data are traversed, the more effective tiling is. Long average transaction length in the transactional database could be an indication of frequent traversal, as for each items in the candidate set, the database are traversed, and a projected database is constructed for lower-level recursion.

Similar as tiling, compaction optimizes the frequent accesses to dispersed data. It is more effective if the accesses are more frequent.

SIMDization works better on dense database, as more operations can now be done in parallel. Let $T$ be a transactional database over itemset $I$. Let the total number of item occurrences in $T$ be $N$. The density of the database is
defined as \( \frac{N}{|T||T'|} \), which is the percentage of the number of 1s in the dense representation shown in Figure 2.10.

### 2.6.2 Selecting the optimal set of optimizations

We have some understanding on how inputs affects the individual optimization performance, the problem of how to systematically choose the group of optimizations to achieve the best performance, however, remains untouched.

The problem of algorithm selection has been named in [Ric76]. The basic idea is that the algorithms provide a classification of inputs, with each class consisting of the set of inputs for which a particular algorithm is better. One wishes to find a classifier that associates (with high probability) each input to the correct class. One approach that has been studied by several authors [LBNA+03, LGP05, TTT+05] is to use machine learning algorithms for this purpose: A training set of inputs are selected; the algorithms are run on these inputs and each input is labeled with the best algorithm; the classifier is trained to identify the labels; the resulting classifier is then used during execution to select which algorithm to run.

Two choices are critical to the success of this approach: (1) one needs to select the set of features that are used to classify inputs; this feature set should be easy to evaluate and should be sufficient to distinguish inputs that have different labels; and (2) one needs to use a training set that is reasonably representative of real inputs. Both are hard problems especially for frequent pattern mining. To solve (1) we need to understand the effectiveness of various optimizations, and find the input characteristics that are more relevant to determine their relative speedups. The difficulty for (2) is that there are only a few publicly available real-world databases, and most of them are too small, so synthetically generated databases need to be used during training. However, the databases generated by the currently available generators such as the IBM Quest Dataset Generator are not representative of the real world databases.

We propose an SVM (support vector machine) [Vap95] based learning method to train such a classification system. We took the LCM algorithm and applied various optimizations. We trained our SVM system by running different versions of optimized code on the synthetic databases. Then, we tested the trained SVM system to predict the best group of optimizations on synthetic databases.

### 2.6.3 The support vector machine (SVM)

Support vector machine [Vap95] is a powerful kernel based machine learning algorithm. It has been widely used in many application domains for hard learning problems, such as optical character recognition, text categorization, and biological sequencing, etc. Next we briefly describe the SVM learning algorithm.

Support vector machine is a kernel based learning algorithm. The main idea of kernel based learning is to embed an input space \( S \subseteq X \) into a vector space \( \mathbb{R}^N \), of high dimensionality. After that linear algorithms, that are efficient and well understood, can be used for classification and regression. Figure 2.20(a) shows that two linearly non-separable
classes become linearly separable after embedding the points from a two dimensional space into a three dimensional space. The embedding mapping is often denoted by $\phi : X \rightarrow \mathbb{R}^N$.

We do not need to perform the embedding explicitly as long as we can compute the pairwise inner products of the image vectors of any pair of data points. We assume that a kernel function $K(x, y) = \langle \phi(x), \phi(y) \rangle$ is available to perform this calculation.

Consider a binary classification problem. A support vector machine embeds the input data points into a high dimensional feature space and then searches for a separating hyperplane that maximizes the minimum distance from any point to the hyperplane, as illustrated in Figure 2.20(b).

After an SVM based classification system is obtained, it can be used to predict the class of a test point by calculating on which side of the hyperplane a point lies. The framework can be easily extended to classifiers with more than two classes, using multiple separating hyperplanes.

Different kernel functions can be plugged into the SVMs framework in a modular manner. Table 2.11 shows some commonly used kernel functions by SVM. Customized kernel functions can also be used to incorporate domain-

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>$K(x_i, x_j) = x_i^T x_j$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma &gt; 0$</td>
</tr>
<tr>
<td>Radial Basis</td>
<td>$K(x_i, x_j) = e^{-\gamma |x_i - x_j|^2}, \gamma &gt; 0$</td>
</tr>
</tbody>
</table>

Table 2.11: Some commonly used SVM kernels
2.6.4 The algorithm prediction framework

Figure 2.21 depicts the components in our SVM based optimal code prediction system, which can be divided into two stages: the training stage and the execution stage. In the training stage, a synthetic database generator is used to randomly generate synthetic databases. The optimized codes are empirically evaluated by running them on the synthetic databases with different support thresholds. Each input \( d = (T, \xi) \), which consists of a data set \( T \) and a support threshold \( \xi \), is represented by a set of feature values \( x \in \mathbb{R}^N \) and labeled with the best code found for it during the empirical evaluation. These labeled training points are input to the SVM learning module to train an SVM model.

During the execution stage, the feature values \( x' \in \mathbb{R}^N \) of an input \( (T', \xi') \) are extracted at runtime. The SVM model produced in the training stage is consulted to predict the code with the optimal group of optimizations for the input based on its feature values \( x' \). The predicted code is then invoked to perform the actual mining task.

2.6.5 Feature selection

Selecting the right features for the learning module is critical for the system to predict accurately. In fact, we have spent a lot of time searching the appropriate features to differentiate the various versions of optimized codes. Before we explain our selected features, it is important to notice two issues. The first is that the operation needed to extract the feature values from a given database must be computationally cheap. If the feature values are too expensive to compute, the benefit of accurate algorithm prediction will be offset by the added run-time overhead. This requirement precludes the use of features directly related to the actual mining results. The second issue is that the features should be extracted after filtering out all the infrequent items from the database, rather than using the original input database. The reason is that all frequent pattern mining algorithms filter out infrequent items before starting the mining process.
Feature symbol | Description
--- | ---
$N$ | Total number of items, i.e., the number of 1s in the dense representation in Figure 2.10.
$s$ | Similarity of transactions, obtained by average-linkage hierarchical agglomerative clustering [Jia07].
$d$ | Density, defined as $\frac{N}{|T||I|}$. See Section 2.6.1.
$\xi\%$ | Support threshold percentage, defined as $\frac{\xi}{|T|}$.
$h$ | Search depth, defined as $1 - \frac{\xi\%}{d}$.
$l$ | Average transaction length, defined as $\frac{N}{|T|}$.
$f_1 \ldots f_{20}$ | Frequency percentage for most frequent 20 items.

Table 2.12: The selected features. $T$ denotes the transactional database over itemset $I$. $|T|$ and $|I|$ denotes the number of transactions and the number of different items respectively.

The filtering process typically counts for a small portion of the total execution time, and as a result, filtered database’s features capture more accurately the input characteristics determining algorithm performance. Thus, from now on in this section the input database will refer to the filtered input database.

To predict accurately, we had to thoroughly study every aspect of the optimizations and the implementation details of them to search for features that best differentiate their performance. Section 2.6.1 suggests that we should focus on features that estimate the size of the problem, the frequencies of items, the transaction length.

The 26 features selected are defined in Table 2.12. They are $N$, $s$, $d$, $\xi\%$, $h$, $l$, $f_1$, $f_2$, $\ldots$, $f_{20}$. We believe the $N$, $s$, $d$, $\xi\%$, $h$, and $l$ are indications of the problem size. The problem size tends to be big when (1) there are large amount of 1s so that we have more items to process; (2) the transactions are similar so that the traversal tree is likely to be deep; (3) the data are dense so that the number of 1s is relatively great; (4) the support threshold percentage is low, then we have a larger number of mining results; (5) the search is deep; or (6) the transactions are long, implying a larger number of mining results.

Similarity $s$ determines how quickly the support of the itemsets decrease in the subset lattice. The more similar the transactions are, the more slowly the support values decrease in the search space. Search depth $d$ tells how much room is available for the support to decrease from the average item support values to the support threshold. The higher this value is, the more room there is for the support to decrease. Therefore, the smaller depth and similarity are, the more shallow the search space is. The first 6 features in Table 2.12, are not exhaustive measurements of the problem...
size, neither are they independent. Change of one of the features might have affect on another.

$l$ is a good measurement of the breadth of the actual mining tree. The breadth determines how frequently the same (projected) databases are traversed; to what extent the accesses can be tiled. In some algorithms, the $l$ is also related to the length of the linked list in the data representation, e.g., FP-Growth.

$f_1, \ldots, f_{20}$ are the frequencies for the most frequent 20 items. As lexicographic ordering optimizes mostly accesses to frequent items, these twenty features gives information on how many accesses are optimized.

The overhead for extracting features out of the inputs is negligible and is discussed in [Jia07].

### 2.6.6 Case study: selecting the best group of optimizations for LCM

As shown in Figure 2.19(a) and Figure 2.19(b), the group of optimizations that yields the best performance is not always the one that includes all of the possible optimizations. For example, in Figure 2.19(a) database DS3, the combination of lexicographic ordering and tiling seems to be the optimal, outperforming the all bar by about 5%.

We choose LCM as our case study, because it exhibits optimization selection problem aforementioned. Another reason is that LCM is the best implementation of frequent pattern mining that is publicly available. Four categories of optimizations can be used in LCM, they are lexicographic ordering, software prefetch, data structuring patterns including compaction and aggregation, and tiling. Although there are totally 16 versions of code with different combinations of optimizations, we choose three among them to study the optimization selection problem. The reason behind it is that (1) some versions of code never performs the best; (2) performance of some code are quite similar to another.

The three codes that we choose are the original code with no optimization, the code with tiling optimization and the code with all optimizations enabled (called all). We use original, tile and all to refer to these three codes below. The performance of these three codes are quite different from each other.

**Rationale behind the selected features**

In section 2.6.1, we have discussed some qualitative characteristics of the input database that will determine the performance of a particular optimization. In this section we relate those characteristics to the selected features (see Section 2.6.5).

Generally, tile and all outperform original. Both tile and all, however, use additional data structure for optimization. For given inputs, they take more memory than original. Therefore, the original tends to perform better when the problem size is so big that it does not fit the physical memory.

From experimental results, we found that the software prefetch and data structuring pattern do not contribute much to the overall speedup. The performance difference between tile and all are mainly determined by the effectiveness of lexicographic ordering. Keep in mind, the tile performs best when tiling is effective and lexicographic ordering is not.
Table 2.13: The three codes’ favorite area in the feature space.

This implies a great average transaction length and low frequencies for most frequent items.

The all performs best when both tiling and lexicographic ordering are working effectively. This require a moderate average transaction length and high frequencies for most frequent items.

Table 2.13 shows each code’s favorite area in the feature space.

### 2.6.7 Experimental results

In this section we describe the experiments we conducted to test the effectiveness of our prediction system. Section 2.6.7 discusses our experimental setup, and presents the results obtained. The database generator that we used are from [Jia07].

**Experimental Setup**

**Training set generation:** We generated the synthetic databases using the modified IBM generator described in [Jia07]. The input parameters to the generator, such as “number of transactions”, “number of items”, “average transaction length”, are randomly chosen within the range of the values in the real world databases. Other parameters such as “average pattern length”, “confidence of patterns” and “correlation between patterns” are randomly generated from an arbitrary range. The item frequency distribution is randomly picked from “Gaussian”, “Zipf”, “Poisson”, “Uniform”, “Exponential” and “Real” distributions. The “Real” distribution simulates the item frequency distribution of the “chess”, “connect”, “accidents”, “mushroom”, “pumsb”, “pumsb_star” or “retail” databases with equal probability, using the kernel density estimator.

**Performance evaluation:** We collected performance data for the three codes on 1000 data points. The experiments are done on machine M2 as shown in Table 2.9. Although the machine has a dual core, we execute the codes one by one to avoid possible interference. The maximum allowed execution time for each run is 350 seconds. If the algorithm does not complete within that period, the process is terminated and the maximum execution time is recorded as the total execution time. We use `time` command to get the running time.
**Code selection:** The codes that we selected are optimizations on the LCM, the best implementation from the FIMI 2004 workshop [JGZ04]. The three codes selected are original, tile and all. These are explained in Section 2.6.6.

**Training points selection:** Although we generated a total of 1,000 synthetic inputs, not all of them were included as training points. The reason is that some inputs are more useful than others for training. For example, some inputs are too trivial for the three algorithm since they all complete in less than 1 second. Some inputs are too hard for all of them and none of them completes within the given maximum period. Obviously, these two kinds of inputs are of little value for training. The operating system might introduce performance noise into the evaluations by context-switching, page-swapping etc. Due to these considerations, we removed inputs such that none of the codes terminated within 350 seconds. We also removed inputs such that the running time for the fastest code is less than 1 second. Left are 455 data points, among which 346 points are randomly selected as the training set and 109 points are selected as the test set. We trained our SVM based classification system using the training set. Then, we tested the trained SVM system on the test set.

**SVM learning module:** In our system, we use the popular SVM library libsvm [CL01] as the learning module. We choose to use the RBF kernel, because it offers non-linear learning capability and has fewer parameters to tune than the polynomial kernel. We directly take advantage of libsvm’s multi-class classification functionality to predict on the three algorithms.

**Prediction Results**

For the 455 points, Figure 2.22 gives the number of times that each code wins. The all wins for most of the time (363 times). The tile wins for 76 cases. The original wins for the fewest number of times, only 16 instances. This is consistent with our expectation. As the all uses all applicable optimizations, each of which contributes to some of the performance improvement, the all outperforms the other two codes in most of the cases. Since the original has no architectural optimization at all, it performs poorly in most of the cases.

Figure 2.23 shows the prediction result for the 109 data points in the test set. Starting from the leftmost bar, it is the average execution time for a perfect prediction, which is marked as optimal. The optimal is the theoretical best that one can get. The second bar is the average execution time when using the code predicted by our SVM classification system. It is very close to the perfect prediction. The next three bars are the average execution time for original, tile and all respectively. We can see, the single best code, all, is about 10% slower than our prediction. Overall, our prediction is effective and the result is near optimal.

To better understand the impact of the selected features on the performance of the mining algorithms, we select three inputs that are illustrative points for which each code version performs best. The execution time for these three data points in shown in Table 2.14. Table 2.15 lists the feature values for these three points.
In all the three points selected, the fastest algorithm significantly outperforms the second fastest. The SVM system correctly predicts on these important inputs, although it mispredicts on some inputs where the penalty of misprediction is relatively small. This misprediction is mainly due to our training point selection strategy described in Section 2.6.7 that emphasizes prediction accuracy on important tasks.

The feature values in Table 2.15 can explain why each algorithm wins on the three important inputs.

For the first input, both the similarity $s$ and the support percentage $\xi\%$ are high. Also the frequencies $f_1, \ldots, f_{20}$ are high. All these imply a large problem size, which possibly requires a lot of memory. The original wins in this case, because it requires less memory. tile and all can not complete within the maximum execution time due to their

<table>
<thead>
<tr>
<th>Point</th>
<th>Winner</th>
<th>original</th>
<th>tile</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>original</td>
<td>153.090</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>tile</td>
<td>9.065</td>
<td>3.661</td>
<td>5.971</td>
</tr>
<tr>
<td>3</td>
<td>all</td>
<td>91.330</td>
<td>67.948</td>
<td>40.786</td>
</tr>
</tbody>
</table>

Table 2.14: Execution time for three example data points. “–” marks the code that does not terminate within the maximum allowed time (350 seconds).
<table>
<thead>
<tr>
<th>Feature symbol</th>
<th>Point 1</th>
<th>Point 2</th>
<th>Point 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>5124726</td>
<td>5968776</td>
<td>8650312</td>
</tr>
<tr>
<td>$s$</td>
<td>0.6374</td>
<td>0.0655</td>
<td>0.2623</td>
</tr>
<tr>
<td>$d$</td>
<td>0.5291</td>
<td>0.0339</td>
<td>0.2118</td>
</tr>
<tr>
<td>$\xi%$</td>
<td>0.7236</td>
<td>0.0035</td>
<td>0.7325</td>
</tr>
<tr>
<td>$h$</td>
<td>0.1462</td>
<td>0.8968</td>
<td>0.0567</td>
</tr>
<tr>
<td>$l$</td>
<td>30.687</td>
<td>39.7918</td>
<td>34.7402</td>
</tr>
<tr>
<td>$f_1$</td>
<td>0.9799</td>
<td>0.1798</td>
<td>0.552</td>
</tr>
<tr>
<td>$f_2$</td>
<td>0.9587</td>
<td>0.1789</td>
<td>0.5464</td>
</tr>
<tr>
<td>$f_3$</td>
<td>0.951</td>
<td>0.1759</td>
<td>0.5322</td>
</tr>
<tr>
<td>$f_4$</td>
<td>0.9508</td>
<td>0.1697</td>
<td>0.5321</td>
</tr>
<tr>
<td>$f_5$</td>
<td>0.9338</td>
<td>0.1461</td>
<td>0.5296</td>
</tr>
<tr>
<td>$f_6$</td>
<td>0.9331</td>
<td>0.1435</td>
<td>0.5294</td>
</tr>
<tr>
<td>$f_7$</td>
<td>0.931</td>
<td>0.1361</td>
<td>0.5261</td>
</tr>
<tr>
<td>$f_8$</td>
<td>0.9307</td>
<td>0.1349</td>
<td>0.5136</td>
</tr>
<tr>
<td>$f_9$</td>
<td>0.9304</td>
<td>0.1301</td>
<td>0.5111</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>0.9296</td>
<td>0.1292</td>
<td>0.509</td>
</tr>
<tr>
<td>$f_{11}$</td>
<td>0.9295</td>
<td>0.1279</td>
<td>0.5007</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>0.8794</td>
<td>0.126</td>
<td>0.4946</td>
</tr>
<tr>
<td>$f_{13}$</td>
<td>0.8755</td>
<td>0.1249</td>
<td>0.4934</td>
</tr>
<tr>
<td>$f_{14}$</td>
<td>0.8743</td>
<td>0.1231</td>
<td>0.492</td>
</tr>
<tr>
<td>$f_{15}$</td>
<td>0.8614</td>
<td>0.1182</td>
<td>0.4795</td>
</tr>
<tr>
<td>$f_{16}$</td>
<td>0.8304</td>
<td>0.1179</td>
<td>0.4767</td>
</tr>
<tr>
<td>$f_{17}$</td>
<td>0.8281</td>
<td>0.1173</td>
<td>0.4764</td>
</tr>
<tr>
<td>$f_{18}$</td>
<td>0.8084</td>
<td>0.117</td>
<td>0.4749</td>
</tr>
<tr>
<td>$f_{19}$</td>
<td>0.7917</td>
<td>0.1165</td>
<td>0.4704</td>
</tr>
<tr>
<td>$f_{20}$</td>
<td>0.7857</td>
<td>0.116</td>
<td>0.4702</td>
</tr>
</tbody>
</table>

Table 2.15: Feature values for the three data points.
even larger memory requirement.

The average transaction length of the second input is very long. Tiling in this case will have better performance, because the repetitive traversals to the transactional database are more frequent.

In a general case like the third input, the all works best, because all optimizations are effective. Particularly, the \( f_1, \ldots, f_{20} \) are medium.

\section{Related work}

Since the introduction of frequent pattern mining, a large number of algorithms and implementations [AIS93, AS94, GZ01, Goe02, BCG01, SON95, ZPOL97, HPY00, LPWH02, PHL+01, ZG03] have been proposed. Different algorithms and implementations use significantly different data representations and access them differently. Some algorithms adapts algorithm’s data structures and traversing order according to input features [LPWH02, LLY+03, OPPS02, OLP+03]. For example, \textit{OpportuneProject} [LPWH02] dynamically chooses between different data structures and counting methods for the projected transactional database using heuristics that estimate the database density. \textit{AFOPT} [LLY+03] adaptively uses three different structures: arrays, AFOPT-tree and buckets to represent the projected database according to the density of the database. \textit{DCI} [OPPS02] and \textit{kDCI} [OLP+03] deal with database peculiarities by dynamically choosing between distinct optimization strategies. In dense databases, identical sections appearing in several bit-vectors are aggregated and clustered to reduce the number of intersections to be performed. In sparse databases, the runs of zero bits in the bit-vectors are promptly identified and skipped.

Ghoting et al. [GBP+05] have studied the problem of ALSO for some tree-based frequent pattern mining imple-
They proposed cache conscious prefix-tree to improve spatial locality and also enhance the benefits from hardware cache line prefetch. Tiling is used to improve the temporal locality. Targeting SMT processors, a thread-based decomposition is used to ensure cache reuse between threads that are co-scheduled at a fine granularity. We have included some of these optimizations as patterns for completeness, knowing that many of these optimizations are tied to tree based implementations. However, we did not apply them in our evaluation because we wanted to study the impact of the newly proposed patterns. We believe that the new optimizations are complementary to existing ones.

In the database domain, optimizations have been proposed for core database algorithms to improve cache performance [BDFC00, SKN94]. Rao and Ross [RR99, RR00] proposed two new types of data structures: Cache-Sensitive Search Trees and Cache-Sensitive B+ Trees. Studies [CGM01, CGMV02, CAGM04] have shown software prefetch could improve searches on B+ trees and Hash-Join operations. Software jump-pointer prefetch has been proposed and evaluated on intensive pointer benchmarks [RS99], which yields an average speedup of 15%. Ailamaki et al. [ADHW99] examined DBMS performance on modern architectures. They concluded that poor cache utilization is the primary cause of extended query execution time.

Empirical search has been used by library generators to overcome the limitations of compilers to generate efficient code. Examples of well-known library generators include PHiPAC [BAwCD97] and ATLAS [WPD01] for linear algebra, and FFTW [FJ05] and SPIRAL [PMJ+05] for discrete transforms. During the installation of the library, these generators produce different versions of the algorithm they implement. These versions are executed in the target machine and the one that delivers the best performance is selected. In all these libraries, performance is data independent, so that the selection depends only on machine characteristics.

Examples of library generators where the performance of the problem solved depends on the input data characteristics are SPARSITY [IYV04], the adaptive sorting generator [LGP04, LGP05], and the adaptive algorithm selection in STAPL [TTT+05]. SPARSITY generates a sparse matrix-vector multiplication, where parameter values for register blocking are set based on the target machine and the sparsity of the matrix. The adaptive sorting generator and the work in STAPL examine the input characteristics such as number of keys, standard deviation or degree of sortedness to determine the best sorting algorithm. In all these cases the authors identify simple input features to drive the algorithm or implementation selection. A difference between these works and the work presented here, is that sorting or sparse matrix-multiplication are simpler than frequent item mining, and as a result it is easier to identify the input features that determine the performance difference.
Chapter 3

The near-memory processor

Many data-intensive applications, including several key ones from the defense domain, are not supported efficiently by current commodity processors. These applications often exhibit access patterns that, rather than reusing the data, stream over large data structures. As a result, they make poor use of the caches and place high-bandwidth demands on the main memory system, which is one of the most expensive components of high-end systems.

In addition, these applications often perform sophisticated bit manipulation operations. For example, bit permutations are used in cryptographic applications [Sch95]. Since commodity processors do not have direct support for these operations, they are performed in software through libraries, which are typically slow.

To address this problem, we propose the use of a heterogeneous architecture that couples on one chip a commodity microprocessor together with a coprocessor that is designed to run well applications that have poor locality or that require bit manipulations. The coprocessor supports vector, streaming, and bit-manipulation computation. The coprocessor is a blocked-multithreaded narrow in-order core. It has no caches but has exposed, explicitly addressed fast storage. A common set of primitives supports the use of this storage both for stream buffers and for vector registers.

To assess the potential of the NMP, we simulate a state-of-art machine with an NMP in its memory controller. We use a set of 10 benchmark and kernel codes that are representatives of applications we expect to use the NMP for. The focus in this evaluation is on multimedia streaming applications, encryption and bit processing. We find that these codes run much faster on the NMP than on an aggressive conventional processor. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8.

3.1 Background and motivation

High memory latency is a major performance impediment for many applications in current architectures. In order to hide this latency, one needs to support a large number of concurrent memory accesses, and to reuse data as much as possible once brought from memory.

Vector processing is a traditional mechanism used for latency hiding. Vector loads and stores effect a large number of concurrent memory accesses, possibly bypassing the cache. With scatter/gather, the locations accessed can be at
random locations in memory. Vector registers provide the large amount of buffering needed for these many concurrent memory accesses. In addition, vector operations can use efficiently a large number of arithmetic units, while requiring only a small number of instruction issues, a simpler resource allocator, less dependency tracking and a simpler communication pattern from registers to arithmetic units.

The vector programming paradigm is well understood and well supported by compilers. It works well in applications with a regular control flow that fits the data parallel model [Rus78].

A more general method to hide memory latency is to use multithreading, supporting the execution of multiple threads in the same processor core, so that when one thread stalls waiting for memory, another one can make progress [URŠ03]. One very simple implementation is the use of blocked multithreading that involves running a single thread at a time, and only preempting the thread when it encounters a long-latency operation, such as an L2 cache miss or a busy lock. This approach was implemented in the Alewife [ABC+95] and the IBM RS64IV [BEKK00]. It has been shown that blocked multithreading can run efficiently with only a few threads or contexts [WG89].

When multithreading is used, it is very desirable to provide efficient inter-thread communication and synchronization mechanisms between the threads. Producer-consumer primitives are particularly powerful. With these, one can very efficiently support a streaming programming model [DHE+03, KRD+03, KDR+01]. A stream program consists of a set of computation kernels that communicate with each other, producing and consuming elements from streams of data. This model suits data intensive applications with regular communication patterns, like many of the applications considered in this chapter.

When the stream model is used, one obtains additional locality by ensuring that data produced by one kernel and consumed by another are not stored back to memory. Stream architectures such as the Merrimac [DHE+03] do so by having on-chip addressable stream buffers, and managing the allocation of space in these buffers and the scheduling of producers and consumers in software. The compiler needs to interleave the execution of the various kernels, a task that is not done efficiently by present compilers [Han]. Alternatively, one can use blocked multithreading and suitable hardware supported synchronization to ensure that the producer is automatically descheduled and the consumer is scheduled when data has been produced and is ready to be consumed. This leads to a simpler target model for compilers, as they compile sequential threads and synchronization operations between threads. This design also handles better tasks with nondeterministic execution time.
3.2 Important concepts

Several important concepts are related to the NMP. We review these concepts, in particular, vector, streaming, and bit manipulation, in this section. We briefly explain the concepts of vector processing in Section 3.2.1. In Section 3.2.2, we introduce the streaming processors. We survey the bit manipulation instructions in Section 3.2.3.

3.2.1 Vector architecture

Vector processors are commercialized long before the superscalar processors. One of the best known and most successful vector processor, the Cray-1 [Rus78], dates back to 1970s. In contrast to the later successful instruction-level parallelism (ILP) machines, the vector processors take a different data-level parallelism (DLP) approach. (1) One vector instruction specifies a great deal of work – the operations that are applied to vectors, which is linear arrays of data. As much fewer instructions are needed to describe a computation task, there are much fewer instructions in-flight during the execution. The control in the processor is therefore much simpler. (2) The operations on vector elements are independent on each other, there are much fewer hazards for the hardware to check, also resulting in a simpler control logic. (3) With memory access patterns implied in the instruction, the memory accesses could be easily pipelined.

A vector code example

Figure 3.1(a) shows the operation for a scalar instruction add R3, R1, R2. The source operands are stored in R1 and R2. R1 and R2 are added together and the result is stored in the destination register R3. For a vector instruction shown in Figure 3.1(b), the add operations are applied to all the elements in the source vector registers and the result is stored in V3. For a regular vector instruction, the memory accesses to vectors are always sequential or strided, which can be efficiently scheduled.
Table 3.1: Scalar and vector code example

Table 3.1 gives a comparison between a scalar and the correspondent vector code. The code is to add two 64-element arrays, a and b, and store the result to an array c. Each element in the arrays has 64 bits, i.e., 8 bytes.

In the scalar code in Table 3.1, a loop is used to go over all the elements in the arrays. For each pair of elements in the source array, after the addition ADD.D, the offset registers R1, R2 and R3 are incremented, and the result is stored back to memory. Also the iteration counter in R4 are decremented.

Assuming the max vector length is greater than 64, the vector code in Table 3.1 is simpler. Only 5 instructions are needed. LI VLR, 64 is to set the vector length to 64. ADDV.D V3, V1, V2 performs the add operation on all of the elements in V1 and V2.

<table>
<thead>
<tr>
<th>C code</th>
<th>Scalar code</th>
<th>Vector code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; 64; i++) c[i] = a[i] + b[i];</td>
<td>LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</td>
<td>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of scalar code and vector code example in Table 3.1

<table>
<thead>
<tr>
<th></th>
<th>Scalar code</th>
<th>Vector code</th>
</tr>
</thead>
<tbody>
<tr>
<td># instructions fetched</td>
<td>$1 + 9 \times 64 = 577$</td>
<td>5</td>
</tr>
<tr>
<td># operations executed</td>
<td>$1 + 9 \times 64 = 577$</td>
<td>$1 + 4 \times 64 = 257$</td>
</tr>
<tr>
<td># loop overhead</td>
<td>$5 \times 64 = 320$</td>
<td>0</td>
</tr>
<tr>
<td># branches</td>
<td>64</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 3.2 compares the scalar code and the vector code in Table 3.1. In the scalar code, great overhead is associated with the loop – 320 instructions total. The overhead includes the instructions to update the offsets, instruction to decrement the iteration counter and the conditional branch. These instructions need to be fetched, decoded and executed. The control hazard in the scalar code is great too – 64 branches total. For the vector version of the code, much fewer instructions are used.

**An example vector architecture**

Figure 3.2 shows the basic structure of a vector-register architecture, VMIPS [HP02], which is loosely based on the Cray-1 [Rus78]. This processor has a scalar architecture just like MIPS [Pri95]. There are eight 64-element vector registers, and all the functional units are vector functional units. These functional units are fully pipelined. Vector instructions are defined both for arithmetic and for memory accesses. Each vector register in VMIPS contains 64 elements. The elements in a vector register are always accessed together in vector instructions. Each vector register has three ports – two read ports and one write port, to allow high degree of overlap among vector operations. Note that the scalar registers are also connected with the vector functional units to perform vector-scalar operations. They are also connected with the load-store unit to provide data to compute addresses. The vector load-store unit loads or
stores vectors to or from the memory. This unit is also fully pipelined.

Actually, the functional units in Figure 3.2 can be pipelined, replicated or both. Figure 3.3 shows the structure of a vector unit containing four lanes [HP02]. The vector registers and functional units are divided across the four lanes. Each lane holds and processes every fourth element. Note that the accesses to a portion of the register file (locations containing every fourth element) are localized in each lane. This dramatically reduces the control complexity of the vector processor.

**Advantages of vector processing**

The vector processing model introduces fewer instruction overheads; moderate number of in-flight instructions are sufficient to exploit the parallelism. The control logic is much simpler than that of a superscalar processor. More functional units could be easily integrated into more lanes to provide higher performance. The memory access patterns of a vector instruction is easy to predict, which enables a possible high performance pipelined memory system.

The power consumption of vector processor is low. As the power consumption is proportional to the square of the frequency, one could double the number of lanes and halve the running frequency to maintain the same performance while consuming only half of the power.

Overall, the vector processor is a compact and scalable design. It has high and predictable performance.
Disadvantages of vector processing

The disadvantages of vector processor are mainly associated with the high cost. As vector processors use non-commodity parts, they are expensive. The highly pipelined independent memory modules are costly. High bandwidth from the memory means more pins off the vector processor. The packaging of the vector processor tends to be hard and expensive.

Another disadvantage of vector processing is the application domain. The vector processor works good on applications with high data level parallelism. For those applications that are hard to be vectorized, vector processing does not help.

Due to the disadvantages mentioned, the vector processors are losing their popularity, while the performance of superscalar processors is catching up. Other competitors are the massive microprocessor clusters. They are cheap alternatives to most of the supercomputing problems.

3.2.2 Streaming processors

The Imagine processor [KDR+01] might be the first interesting streaming processor. Figure 3.4 shows the structure of Imagine. It provides high performance with 48 floating-point arithmetic units and an area- and power-efficient register organization. A streaming memory system loads and stores streams from memory. A stream register file provides a large amount of on-chip intermediate storage for streams. Eight VLIW arithmetic clusters perform SIMD operations on streams during kernel execution. Kernel execution is sequenced by a micro-controller. A network interface is used to support multi-Imagine systems and I/O transfers. Finally, a stream controller manages the operation of all of these units.

The stream programming model

Applications for Imagine are programmed using the stream programming model [KRD+03, TKA02]. This model consists of streams and kernels. Streams are sequences of similar data records. Kernels are small programs which operate on a set of input streams and produce a set of output streams.

Figure 3.5 shows an example for stream processing. It is taken from [KDR+01]. Table 3.3 shows the pseudocode. The input data are the camera images. They are formatted as streams. The convolution kernels process the input streams and produce filtered streams. The circular arrows in the diagram is the stream of partial sums produced and needed by the same kernel as it processes future rows.

We can see from Figure 3.6, the stream programming model maps directly to the Imagine architecture, i.e., the kernels execute on the arithmetic clusters and streams pass between kernels through the stream register file. The SIMD nature of the arithmetic clusters and compound stream operations enables Imagine to exploit data parallelism.
main function:
1 prime partials with arc_rows[0..5]
2 for (n = 6; n < numRows; n++){
3 src = load( src_rows[n] ); // 'src' stream gets one row of source image
4 convolve7x7( src, oldpartials7, &tmp, &partials7 );
5 convolve3x3( tmp, oldpartials3, &cnv, &partials3 );
6 convolved_rows[n-6 ] = store( cnv ); // store 'cnv' stream to memory
7 swap pointer to start of oldpartials and partials for next time through the loop
8 }
9 drain partials to get convolved_rows[numRows-6..numRows-1]

convolve7x7(in, partials_in, out, partials_out){
...
while (! in.empty()){
    in >> curr[6]; // Input stream element
    // Communicate values to neighboring
    // clusters (edge clusters get buffered data from prev iteration)
    for (i=0; i < 6; i++)
        curr[i] = communicate(curr[6], perm[i])
    for (i = 0; i < 7; i++)
        rowsum[i] = dotproduct(curr, fltr[6-i]);
    partials_in >> p;
    out << p[0] + rowsum[0];
    for (i = 0; i < 5; i++)
        p[i] = p[i+1]+ rowsum [i+1];
    p[5] = r[6];
    partials_out << p;
}
}

Table 3.3: Pseudocode for the convolution stage of stereo depth extraction.
3.2.3 Bit permutation instructions

Bit manipulation applications are frequently appearing in the defense domain. Bit permutation is one of the most important bit manipulation operations. It is used in cryptographic algorithms. Some other manipulations are used in multimedia applications. Bit permutation operations, however, are not supported efficiently in modern architectures. This section surveys the methods to perform arbitrary bit permutations.

The mask-and-shift-or method

This method works on almost any architecture. The trick is to extract bits from the source register, move (shift) it to the new location and deposit it to the destination register. The following code moves the 1st (left most) bit to the 5th bit. The source register is Rs; the destination register is Rd.

1: load R2, 0x80000000
2: and R3, Rs, R2
3: shift_right R4, R3, 4
4: or Rd, R4, Rd

Line 2 extracts the 1st bit out of the Rs. Line 3 shifts the 1st bit to the 5th position. Line 4 puts the bit into the destination register Rd. Totally four instructions are required to permute one bit. For a 64-bit value, this method would take $64 \times 4 = 256$ instructions.
Table lookup

Table lookup is one other popular method to permute bits. Before the permutation takes place, a conversion table is set up for quick lookups. For each possible value of the input, a permuted value is returned as the lookup result. For this method, each different permutation requires a different lookup table, which may take substantial amount of memory. For an \( n \)-bit value, the lookup table has \( 2^n \) entries.

One approach to avoiding extensive memory consumption is to break the \( n \)-bit value to small sections and perform one lookup for each section. Suppose we break the \( n \)-bit value into \( m \) sections, each of which is a \( \frac{n}{m} \)-bit value. We need \( m \) times table lookups to permute an \( n \)-bit value, where each lookup is within a table with \( 2^{\frac{n}{m}} \) entries. The result of each table lookup is an \( n \)-bit value, out of which \( m \) bits are from the selected bits to permute, and \( n - m \) bits are 0s. A final step is to \( \lor \) these \( m \) results to get the final result for the permutation.

For example, to permute a 64-bit value, we break the value into bytes and perform a table lookup for each byte. We have one lookup table for each byte, i.e., totally 8 lookup tables, each of which has \( 2^8 = 256 \) entries. Each entry in the tables contains a 64-bit value, where the 1s in the entry marks the places where the 1s in the source value go in the destination value. The following 23 instructions are needed to permute a 64-bit value, plus instructions to extract the indices from the source register. 8 instructions to load the lookup table indices; 8 instructions to look up the table; 7 instructions to assemble (\( \lor \)) the lookup results.

Lee et al. has pointed out in [LSY01] that the minimal number of MIPS instructions for arbitrary \( n \)-bit permutations with no repetitions is \( \log(n!) \sim n\log(n) \).
Some new permutation instructions

Some new instructions are summarized in [LSY01, HL06]. We introduce PPERM and GRP instructions in this section.

The PPERM $x$, Rs, Rc, Rd instruction does the following. The Rs is the source register; the Rd is the destination register. $x$ is the section in Rd to change. Rc is the configuration register. For the $x$th section in Rd, each bit in the section are generated from the bits in Rs. Positions of these bits in Rs are specified in Rc.

Figure 3.7 [LSY01] shows an example of the PPERM instruction. The shaded bytes in R3 are the bits affected in this instruction. These 8 bits are from the bits in R1, whose positions are specified in R2. The example moves the 2, 14, 22, 8, 32, 37, 44, and 51$^{th}$ bits in R1 to the 8--15$^{th}$ bits in R3.

The GRP Rs, Rc, Rd instruction [SL00] can selectively move bits to the left or the right portion of the word. The Rs is the source register; the Rd is the destination register; the Rc is the configuration register. A 0 bit in Rc causes the correspondent bit in Rs to move to the left group of Rd; the correspondent bit in Rs goes to the right group of Rd otherwise. The relative positions of bits in the left and right groups do not change. Concatenating these two groups gives the result in Rd (See Figure 3.9 [SL00] for an example). $\log_2 n$ steps are sufficient to permute an $n$-bit
Figure 3.7: Diagram of flow of bits for PPERM 1, R1, R2, R3. R2 = 0x020E160820252C33. The numbers 2, 14, 22, 8, 32, 37, 44, and 51 are the bit positions in R1.

value [SL00]. The process is similar like the radix sort.

**Bit matrix multiply**

The *bit matrix multiply* instruction, also known as *bmm*, first appears in Cray machines [cra]. The instruction *bmm* takes two source registers, which are 64 bits and 64 × 64 bits respectively. The 64-bit register stores the value to be transformed, the 64 × 64-bit register stores the configuration. The *bmm* instruction *bit-multiplies* the two source registers and stores the result in a 64-bit destination register.

Figure 3.8 shows an example of the *bmm*. For simplicity, the source registers are 4-bit and 4 × 4-bit. The 4-bit source register is *bit-multiplied* with each row of the 4 × 4-bit register. As marked in the figure, the 4-bit source register 0 1 1 0 is multiplied with the second row of the 4 × 4-bit source register to generate the 2nd bit of the destination register.

The example also shows how to permute bits. In order to switch two bits, one needs to switch the correspondent two rows in the identity matrix and load it to the 4 × 4-bit register as a configuration. A permutation corresponds to a

\[
\begin{bmatrix}
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \times
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & X1+1X0+1X0+0X0
\end{bmatrix}
\]

Figure 3.8: Bit matrix multiply.
permutation matrix (one 1 in each row and column).

### 3.3 Proposed architecture

#### 3.3.1 Rationale

As discussed in the previous section, vector processing is a well understood, easy to implement mechanism for hiding memory latency, with good software support. Streaming architectures provide a more general latency hiding mechanism, at the expense of a more complex programming model, more complex hardware and the requirement for more advanced compiler technology. However, the streaming model fits well streaming applications where a sequence of kernels are pipelined. Support for the streaming model can be simplified if one uses multithreading, since software does not need to handle the interleaved execution of multiple kernels. Multithreading also handles kernels with variable execution time better.

It turns out that a set of common mechanisms can be used to exploit on chip storage both for vector registers and for stream buffers. With such addressable common storage, it is possible to keep a relatively small amount of state for each executing thread, so that context switching is not expensive; blocked multithreading can be implemented at a relatively low cost and can be used efficiently. Thus, we choose to implement the NMP as an engine that combines vector processing, streaming and blocked multithreading. Finally, we added bit manipulation logic to support bit-oriented applications. As it turns out, all these mechanisms are needed to achieve performance on the applications we consider.

The combination of the vector/streaming models and blocked multithreading is attractive, as modest levels of multithreading are sufficient to address the limitations of these models. Specifically, for vector workloads, processor
stalls caused by short vectors or by highly-variable memory access latencies can often be hidden by preempting the current thread and running another one. Similarly, for streaming workloads, processor stalls caused by the imbalance of computation or memory bandwidth between a producer and a consumer kernel can usually be hidden by preempting the fast kernel, and running the slow one.

To be able to exploit data locality, the NMP has a large, high-bandwidth, multi-bank local memory area that it directly manages. We call it the **scratchpad**. To support streaming efficiently, the NMP supports very low overhead producer-consumer synchronization between concurrent threads, using full/empty bits in the scratchpad. The design is similar to the one used by the HEP [Smi81] and MTA machines[ACC+90].

Since the scratchpad is large, it is impractical to save and restore it upon context switch. Thus, the scratchpad is not part of a thread context — the thread context includes only a small number of scalar and control registers. Since threads running on the same NMP can belong to distinct processes, we need to provide address protection in the scratchpad. We do so by using virtual addressing. Although using virtual addresses slightly increases scratchpad access time, the overhead is modest if data is processed using long vectors and stream buffers, as address translation is performed only once per vector or stream buffer access in the scratchpad, assuming vectors and stream buffers can not cross pages. Such virtualization has the added benefit that scratchpad storage associated with threads that are inactive for a long period of time can be lazily paged out (into the main memory) and brought back on demand when needed.

The NMP also includes instructions for bit processing like those of the Cray machines [Rus78]. In particular, it has a bit matrix register that is used for data permuting instructions such as bit matrix multiply. The bit matrix multiply could be used for rows/columns exchange, bits extraction/permuation and parity check, etc.

Overall, the resulting architecture is fairly general and can speed-up many classes of applications. Our evaluation
is focused on vector, streaming and bit manipulation applications, as these are most challenging for a conventional processor.

In the following sections, we overview the design (Section 3.3.2), describe the scratchpad (Section 3.3.3), give some details on the instruction set (Section 3.3.4), and talk about other issues in Section 3.3.5.

### 3.3.2 Overview of the design

Figure 3.10 shows the NMP in a system like the IBM Power 5. The memory controller is on-die. Each memory controller is associated with an NMP. Each processor has a private L1 cache, and a L2 cache shared between the two processors. The L3 cache is off-chip, shared between all processors in the node. The Fabric Controller connects to the interconnection network, through which the NMP could talk to the rest of the system. The NMP can access any memory location in the system.

Figure 3.11 shows the organization of the NMP. The dashed box encloses the NMP. In the figure, the NMP Interface provides an interface for the NMP to communicate with the rest of the system. The main processor(s) communicate with the NMP via the Invocation Register Sets (IRSs) (Section 3.3.5). As soon as a request from a main processor arrives, the Thread Management Unit creates a thread and inserts it into the NMP’s job queue.

Figure 3.12 shows the organization of the NMP core. The core is a low-issue in-order processor. It does not have data caches but, as indicated before, it uses the explicitly-managed fast scratchpad memory (Section 3.3.3). It includes scalar functional units, vector functional units, a set of general-purpose and control registers, and a single Bit Matrix Register (BMR) to permute the bits within a word [cra].

To save space, there is only a single BMR. The BMR is tagged with the owner thread ID and is not saved upon context switch. If the hardware detects that a thread is going to access the BMR with an inconsistent thread ID tag, an
exception occurs. The operating system then saves the BMR contents and loads the BMR for the current thread.

All threads running on the NMP share the scratchpad. In addition, they can access any main memory location in the machine. To access the main memory, an NMP thread uses the same 64-bit virtual addresses as if it runs in the main processor. Accesses of the NMP to the main memory are handled the same way as accesses by the main processor are handled: they are broadcast on the coherence fabric and snooped by caches in the system. The NMP has a TLB to cache address translation entries that are kept coherent with other TLBs in the system.

### 3.3.3 The scratchpad

The scratchpad is an explicitly-managed storage area for frequently-accessed scalars, vectors and stream buffers. The vectors and streams in the scratchpad are stored sequentially. Data can be moved between memory and scratchpad using vector load and store instructions, including strided access and scatter/gather. The vector units process vectors that are contiguous in the scratchpad. One can use masks to selectively perform operations over elements in a vector. (This is similar to the model provided by a vector register). Thus, one can implement the scratchpad using a multi-banked memory with separate lanes from the banks to the vector units, and a barrel shifter to align vectors.

The NMP supports fine grain, cross-thread synchronization. Each addressable location (byte) is associated with three flags, each of which is one-bit: a full/empty bit\([Smi81]\), an error flag bit, and a mask bit. The first one is for fine-grain synchronization: a synchronized read that consumes the data stalls until the bit is on, while a synchronized write stalls until the bit is off. The error flag bit is used for recording the locations suffered exceptions during the execution of a vector operation. Finally, the mask bit is used to mark the elements of a vector that need to be masked off in a vector operation. (Vector architectures store the mask bits in separate registers, so that the same data can be controlled by different mask vectors; we have not found the need for this extra flexibility in the kernels that we have studied so far.)

For reasons explained in the previous section, the scratchpad is accessed using virtual addresses: the storage is divided into pages (these need be of the same size as main storage pages). Threads running on the NMP address the
local scratchpad using a short virtual address, currently set at 20 bits (8 bit page number and 12 bit displacement, assuming 4K-byte page size for the main memory). The NMP has a TLB that holds entries for all the pages present in the scratchpad. A TLB miss causes an exception that blocks a thread and is handled by a main processor.

Threads also access the main memory using regular (64 bit) addresses. TLB entries are also required for main memory addresses. We can use a common TLB or two separate TLBs.

Accesses to the main memory are snooped by the caches of the regular processors, hence are coherent. No snooping occurs when the NMP accesses the local scratchpad.

The main processors can access the scratchpad data (including the additional bit flags), but these accesses are not coherent, and the mapping (e.g., of the extra bits) is not straightforward, so that these accesses normally occur only in supervisory mode (e.g., for paging out a scratchpad page to memory). The normal mode of operation is that the NMP pulls data from memory (or caches) to the scratchpad and pushes it back.

### 3.3.4 Instruction set architecture (ISA)

In this section, we give some details on the NMP instruction set. The full description can be found in [WSTT05a].

The NMP has 32-bit instructions. For our simulations, we use an augmented MIPS instruction set [Pri95]. New addressing modes are added to handle streams and vectors. New op-codes are added for new arithmetic and logical operations, e.g., bmm, leadz, etc. An NMP arithmetic/logic instruction has two source operands and one destination operand. All operands have to be either in the registers or in the scratchpad. Vector or stream instructions are identified by the addressing mode field in the instruction. Load/store instruction moves data between the memory and the registers or the scratchpad. The scratchpad is treated as a register extension. Data movement between the registers and the scratchpad can be done via arithmetic instructions.

**General purpose registers**

A small number (e.g., 32) of 64-bit general purpose scalar registers are available to each thread in the NMP. These registers store scalars and specifiers, where specifiers are used to refer to scalars, vectors, or streams in the scratchpad (See below for specifiers).

**Control registers**

They include Instruction Pointer Register, status registers and some other control registers. The mask registers which are used for conditional vector instructions are not provided. Each element in the vector, however, has an extra bit to hold the mask bit. The execution of a vector instruction takes no effect on the elements whose correspondent mask
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leadz</td>
<td>Count the leading zeros of a scalar.</td>
</tr>
<tr>
<td>Popcnt</td>
<td>Count the number of ones in a scalar.</td>
</tr>
<tr>
<td>Bmm_load</td>
<td>Load the $64 \times 64$-bit matrix from the scratchpad into the BMR. It is a special vector load instruction (A regular vector load instruction transfers data between the scratchpad and the main memory.).</td>
</tr>
<tr>
<td>Bmm</td>
<td>Bit multiply the source operand with the matrix in the BMR. For $bmm(s_i, s_k)$, each bit $j$ of the 64-bit integer result $s_i$, counting from the highest order bit position down to the lowest, is computed thus: $s_{ij} = popcnt(s_k &amp; bmr_j) \pmod{2}$, where $bmr_j$ is the $j$th row of the BMR [cra].</td>
</tr>
<tr>
<td>Sshift</td>
<td>Logic left- or right-shift a block of data, e.g., 128 bytes. The shift can be rotational or not rotational. In the latter case, zeros are shifted into the block.</td>
</tr>
<tr>
<td>Mix</td>
<td>Bit-interleave higher(lower) half of two words.</td>
</tr>
</tbody>
</table>

Table 3.4: Bit manipulation instructions.

bits are 0s. Note, both the vectors and masks are not part of the context, they are stored in program addressable space (scratchpad).

**Bit matrix register**

(BMR) is a $64 \times 64$-bit register. The BMR is used for the $bmm$ instruction [cra] (Section 3.2.3) to permute the bits within a word. To permute a word $W$ in a register, a $bmm$ instruction bit-multiplies the BMR with $W$, the output of which is stored in the destination register. The $bmm$ instruction enables efficient execution of various functions such as bit permutation, bit matrix transposition, column parity calculation, etc.

**Bit manipulation instructions**

Table 3.4 summarizes the bit manipulation instructions that we propose.

**Addressing modes and specifiers**

Storage in the scratchpad can be interpreted to hold scalars, vectors, or stream buffers, i.e., circular buffers holding queues. The interpretation results from the semantics of the instructions used to access the scratchpad and from information stored in registers.
Instructions specify an opcode; the operands size (byte, half-word, word, etc.); the addressing mode (Figure 3.13), and up to three registers. When direct addressing is used, the register contains a scalar operand. When indirect addressing is used, the register contains a specifier for an operand in the scratchpad. Specifically, it can have a scalar specifier, a vector specifier, or a stream buffer specifier. Bits are included in the register to distinguish different specifiers.

Thus, an instruction \texttt{ADD size mode R1 R2 R3} will add two operands specified by \texttt{R1} and \texttt{R2} and will store the result in a location specified by \texttt{R3}. \texttt{size} specifies whether the additions are performed on bytes, half-words, words or double-words; \texttt{mode} specifies whether each operand is a scalar contained in the specified register (direct addressing) or a scalar, vector or stream buffer stored in the scratchpad (indirect addressing); not all possible combinations for the three operands in one instruction are supported. In the indirect addressing mode, the specifiers are stored in the registers, see Figure 3.14.

A few extra bits are needed in the instruction to encode the operand size (currently 5 choices) and the mode (2 choices, direct or indirect). The extra bits required for these fields are obtained by sacrificing some bits from existing fields, e.g., the shift amount, immediate field.

A scalar specifier is a scratchpad address. The specifier may also specify that the access is conditional on the full/empty bit value in the scratchpad (see below). This can be used for thread synchronization.

A vector specifier consists of a vector start address and a vector length (number of operands).

A stream buffer specifier consists of a buffer start address, the buffer length, number of elements to operate in one operation and a pointer to the head of the buffer (for input operands) or to the tail (for output operands). An input operand is dequeued (if the read only flag is 0) from the head of the buffer and the head pointer in the register is updated (or peeked if the read only flag is 1); the thread blocks if the queue is empty. An output operand is enqueued at the tail of the buffer and the tail pointer in the register is updated; the thread blocks if the queue is full. The pointers wrap around at the boundaries of the buffer.

All the specifiers fit in a 64 bit register (remember that addresses have 20 bits).
### Figure 3.14: (1) Vector specifier, (2) Stream buffer consumer specifier and (3) Stream buffer producer specifier.

The three operands of an instruction can be all scalars (from a register, or from the scratchpad); they can all be vectors of the same length (accessed via a vector or stream buffer specifier). One can also mix scalars and vectors as input operands, in which case the scalar is expanded to the vector length.

New instructions are added to move data between memory and scratchpad using strided or indirect vector loads and stores (scatter/gather), as these require more than one register argument.

#### Vector Instructions

Each scalar arithmetic or logic instruction has a vector counterpart which applies the same operation on every element in the vector. Load/store moves data between the memory and the registers or the scratchpad. Sequential, strided and indexed (scatter/gather) main memory access patterns are supported. Vector element size can be smaller than one word so that vector loads and stores can be used for subword operations.

If the operands are vectors then the operation uses the mask bits associated with its input operands in the scratchpad storage, and may set the error bits and mask bits associated with its output operand in the scratchpad.

#### Operations on Stream Buffers

No special instructions are provided to operate stream buffers. Instructions use indirect mode to refer to stream buffers. The specifiers for the stream buffers are updated implicitly when elements enter or leave the buffer. Full/Empty bits are used in the buffer to detect if the buffer is full or empty. If the head of the buffer is empty, the consumer has to...
wait. The producer has to wait until the tail of the buffer is empty to deposit new elements.

**Full/empty bits**

The full/empty bits in the scratchpad are used to avoid underflow and overflow: a consumer marks the element as empty and a producer marks the element as full. Note that the head (resp. tail) of the queue is stored in a register of the consumer (resp. producer), and is not shared; the full/empty bits are in the scratchpad and are shared (a stream buffer is stored in a page that is accessible both by producer and consumer). The current design does not directly support multiple producers or multiple consumers; an additional multiplexing thread is needed to support such. This limitation has not proven a problem with the kernels considered so far.

The scratchpad supports six access types: load, load-if-full, load-if-full-and-mark-empty, store, store-if-empty and store-if-empty-and-mark-full. These access types can be specified explicitly by a scalar specifier; buffer specifiers implicitly require the use of load-if-full-and-mark-empty (for inputs) or store-if-empty-and-mark-full (for outputs). The logic to update the head or the tail of a stream buffers, use mask bits or set error bits is in the functional units.

### 3.3.5 Other issues

**Coprocessor Interface**

The NMP works coupled with the main processor, using a mode where the main processor is the master and the NMP is the slave. The main processor triggers an NMP computation by storing an Invocation Packet into one of the Invocation Register Sets of of Figure 3.11, which are memory mapped in the main processor’s address space. The mapping into user space is done in supervisor mode, while the storing Invocation Packet operation is done in user mode, without a system call. The packet is moved immediately into a queue, clearing the register for a new invocation from the same process (an exception occurs if the queue is full). The invocation packet includes a pointer to the function to invoke, a pointer to its arguments (including a pointer to a completion flag currently set to zero). The main processor can then regularly poll the completion flag. The NMP signals completion by setting the completion flag. We expect this interface to have very low overhead.

**Protection and Virtualization**

An NMP may be executing threads on behalf of more than one process running on the main processor(s). These threads need to be protected from each other. Some NMP threads may even belong to processes that are not running in the main processor(s) but are still alive. In order to use NMP resources efficiently, such threads need to be descheduled.

To do so, we manage NMP contexts as memory, piggy-backing on the virtual memory management infrastructure. Specifically, scratchpad space is allocated in swappable pages; each NMP thread is associated with some “low core”
scratchpad space that is used to save the thread context. The scratchpad pages are paged to main memory by an external pager when physical scratchpad space needs to be allocated to a newly-invoked thread. The paging mechanism ensures that a thread cannot overwrite scratchpad space or memory used by other threads. However, partial sharing of the scratchpad space, via stream buffers, is also possible.

**Exceptions and Context Switching**

A thread may get blocked in the middle of executing a vector computation. The NMP is designed to be able to continue a vector operation from the point where it was stopped. This is the same approach as used in [Koz99]. The same logic is used to handle virtual memory exceptions that happen during the execution of vector loads/stores that move data between memory and the scratchpad.

The handling of vector arithmetic exceptions is postponed until the completion of the vector instruction [Asa98]. The faulting elements are marked in the error flag bits of the destination vector.

### 3.4 Programming model

#### 3.4.1 Processor-NMP communication

Threads are created by processes running on a main processor using a system call that returns a handle – effectively a pointer to an Invocation Register Set; the call fails if no Invocation Register Set is available. Another system call can be used to kill the thread and free the handle.

The communication model between processor and coprocessor is that of an asynchronous procedure call: code running on the main processor can invoke a function on the coprocessor; the invocation specifies the NMP to run this function, a pointer to the function, and arguments. Normally, one of the argument will be a location of a flag to be set by the invoked function upon completion. Thus, the main processor can poll for invocation completion or block until completion.

#### 3.4.2 API for the NMP

The NMP has a simple thread-like API. Initially, a main processor calls `NMP.Connect(Addr)` to establish a connection with an NMP. This is a system call that allocates an Invocation Register Set to the caller process. On success, `NMP.Connect()` returns a handle `N` through which the main processor can have user-level communication to an NMP. The NMP selected by `NMP.Connect()` is the one whose local memory module contains the physical address corresponding to the virtual address `Addr`. From now on, the main processor can use the handle `N` to spawn threads on the NMP that is “near” to the data at address `Addr`. 77
The main processor can create multiple threads over time on the NMP without intervention by the operating system. A thread is created by calling Memthread_Create(Function, Arguments, CompletionFlag, N), where Function is a pointer to the function to execute, Arguments are the functions arguments, CompletionFlag is a simple flag, and N is the handle returned from the previous call. This code writes an Invocation Packet to the Invocation Register Set. The NMP indirectly obtains a precompiled thread from the location in memory indicated by the pointer Function. This is a user-level invocation. The Memthread_Create() invocation returns a status indicating whether the operation succeeded.

The Memthread_Create() invocation is nonblocking: the main processor resumes execution without waiting for the NMP thread to complete. The main processor may check at a later point if the NMP thread completed by calling Memthread_Wait(CompletionFlag), which blocks the calling thread until the CompletionFlag is set. It can also invoke function Memthread_Poll(CompletionFlag) and Memthread_Select(CompletionFlag), whose semantics are the same as poll() and select() in UNIX systems.

Upon completion, the NMP thread sets the completion flag CompletionFlag in memory. Finally, to disconnect from the NMP, the main processor calls the NMP_Disconnect(N) system call. The parameter N is the handle of the NMP to be disconnected.

### 3.4.3 Thread scheduling

A thread executes only one function at a time, picking from its queue a new invocation to execute when the previous one has completed. The NMP hardware schedules runnable threads round-robin. A running thread executes until it exits, or blocks on a synchronization, or idles on a high latency memory access, at which point it is descheduled.

The hardware does not prevent livelock or deadlock; this is the programmer’s responsibility. The hardware however maintains sufficient state so as to allow livelock or deadlock detection by the system, e.g., the time of the last execution by a thread and the last instruction executed.

### 3.4.4 Compilation and run-time

Our current programming model uses library calls for thread creation, thread termination and thread synchronization and a compiler to generates thread code. The run-time supports the allocation and deallocation of thread structures and scratchpad space, while thread synchronization is directly supported by hardware. We have not yet developed a compiler for the NMP new instructions and addressing mode; currently, we insert additional instructions and vector code manually in the source code. The compilation of thread code from high level language requires added support for vectorization and for the generation of bit manipulation instructions; this does not require new compiler technology as such capability has been available in commercial compilers for a long time.
3.5 Evaluation

3.5.1 Evaluation methodology

To evaluate the NMP concept, we use an execution-driven simulator [gro] with a detailed model of the main processor, the coprocessor and the memory system. We model the architecture shown in Figure 3.15, which contains a single main processor and a single NMP. The main processor is a 4-issue out-of-order superscalar with two levels of caches, while the NMP is a 2-issue in-order blocked-multithreaded processor. Main processor, memory controller, and NMP share the same processor chip. The parameters of the architecture modeled are shown in Table 3.5, Table 3.6 and Table 3.7. Note that the main processor has an aggressive 16-stream hardware stride prefetcher. The prefetcher is similar to the one in [PK94], with support for 16 streams and non-unit stride. The prefetcher brings data into a buffer that sits between the L2 and main memory.

For the evaluation, we select a number of small applications that we list in Table 3.8. On average, the applications have 730 lines of C code. The table shows if the applications can be vectorized, use streams, or use bit manipulation instructions. The table also shows the number of concurrent NMP threads used for each application.

The simulated NMP has a MIPS-like instruction set, augmented with vector, streaming, and bit manipulation instructions. Since we do not have a compiler that generates vector or stream codes, we hand-coded the vector, streaming, and bit manipulation instructions. These new instructions are captured and simulated by the simulator. All programs are compiled using GCC compiler version 3.2.1.

To understand the applications, we briefly describe what they do.

*Rgb2yuv* converts an image (1000 × 200 pixels) in RGB color format to YUV color format. We execute four
### NMP Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>4GHz in-order</td>
</tr>
<tr>
<td>Issue Width</td>
<td>2</td>
</tr>
<tr>
<td># Scalar FUs</td>
<td>1Int FU, 1FP FU</td>
</tr>
<tr>
<td># Vector FUs</td>
<td>1Int FU, 1FP FU</td>
</tr>
<tr>
<td># Lanes</td>
<td>16</td>
</tr>
<tr>
<td># Pending Memory Ops (Ld, St)</td>
<td>128, 128</td>
</tr>
<tr>
<td># Contexts</td>
<td>4</td>
</tr>
<tr>
<td>Time to Context Switch</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Policy for Context Switch</td>
<td>Switch after 20 idle cycles</td>
</tr>
</tbody>
</table>

Table 3.5: Parameters of the NMP.

### Memory Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1, L2, scratchpad size</td>
<td>32KB, 1MB, 64KB</td>
</tr>
<tr>
<td>L1, L2 associativity</td>
<td>2-way, 4-way</td>
</tr>
<tr>
<td>L1, L2 line size</td>
<td>64B, 64B</td>
</tr>
<tr>
<td>Main proc. to L1, L2, memory round-trip latency</td>
<td>2, 10, 500 cycles</td>
</tr>
<tr>
<td>NMP to Scratchpad, memory latency</td>
<td>6, 470 cycles</td>
</tr>
<tr>
<td>Bandwidth b.t. vec. units and Scratchpad, Scratchpad and memory</td>
<td>256GB/s, 32GB/s</td>
</tr>
</tbody>
</table>

Table 3.6: Parameters of the memory hierarchy.
### Main Processor Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>4GHz out-of-order</td>
</tr>
<tr>
<td>Fetch Width</td>
<td>8</td>
</tr>
<tr>
<td>Issue Width</td>
<td>4</td>
</tr>
<tr>
<td>Retire Width</td>
<td>8</td>
</tr>
<tr>
<td>ROB size</td>
<td>152</td>
</tr>
<tr>
<td>I-window size</td>
<td>80</td>
</tr>
<tr>
<td>Int FUs</td>
<td>3</td>
</tr>
<tr>
<td>FP FUs</td>
<td>3</td>
</tr>
<tr>
<td>Mem FUs</td>
<td>3</td>
</tr>
<tr>
<td>Pending Ld/St</td>
<td>16, 16</td>
</tr>
<tr>
<td>Branch Pred.</td>
<td>Like Alpha 21464</td>
</tr>
<tr>
<td>Branch Penalty</td>
<td>14 cycles</td>
</tr>
<tr>
<td>Hardware Prefetcher</td>
<td>16-stream stride</td>
</tr>
<tr>
<td>Prefetch Buffer</td>
<td>16KB</td>
</tr>
<tr>
<td>Pref. Buf. Hit Delay</td>
<td>8 cycles</td>
</tr>
</tbody>
</table>

Table 3.7: Parameters of the main processor.

<table>
<thead>
<tr>
<th>Application</th>
<th>Vector?</th>
<th>Stream?</th>
<th>Bit Manip?</th>
<th># Threads</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rgb2yuv</td>
<td>X</td>
<td></td>
<td></td>
<td>4</td>
<td>Convert the RGB presentation to YUV</td>
</tr>
<tr>
<td>ConvEnc</td>
<td>X</td>
<td>X</td>
<td></td>
<td>3</td>
<td>Convolutional encoder</td>
</tr>
<tr>
<td>BMT</td>
<td></td>
<td>X</td>
<td></td>
<td>4</td>
<td>Bit matrix transposition</td>
</tr>
<tr>
<td>BSM</td>
<td></td>
<td>X</td>
<td></td>
<td>3</td>
<td>Bit stream manipulation</td>
</tr>
<tr>
<td>3DES</td>
<td>X</td>
<td></td>
<td></td>
<td>4</td>
<td>3DES encryption</td>
</tr>
<tr>
<td>PartRadio</td>
<td>X</td>
<td>X</td>
<td></td>
<td>3</td>
<td>Partial radio station</td>
</tr>
<tr>
<td>Stream</td>
<td>X</td>
<td></td>
<td></td>
<td>4</td>
<td>Simple vector operations</td>
</tr>
</tbody>
</table>

Table 3.8: Applications evaluated.
threads concurrently. Each thread processes part of the input data stream.

*ConvEnc* performs a convolutional encoder algorithm, which adds redundancy to a binary stream for forward error correction. A binary, half rate (2 bits of output for each input bit) bit stream is encoded with a constraint length of 3. The generating polynomials that we use are \( G_0 = 1 + D_1 + D_2 \) and \( G_1 = 1 + D_2 \). The input stream is 375K bytes.

For *ConvEnc*, we generate three threads: one thread reads the binary stream to be encoded into a stream buffer in the scratchpad; a second thread performs the encoding, processing a 64-byte block at a time, and stores the results into another stream buffer; the third thread takes the results from the stream buffer and writes them back into the memory. We use vector operations, a block shift instruction *Sshift*, and a bit manipulation instruction *Mix*.

*BMT* tests the bit manipulation ability of the NMP. The input is a binary stream (about 4M bits). Each consecutive 1024 × 1024 bits in the stream are treated as a bit matrix. The bit matrices are transposed and the resulting matrices are stored back to the memory. We use four threads, each of which works on a partition of the input data. In each thread, the 1024 × 1024 bit matrices are divided into 64 × 64 submatrices, and the Bmm instruction is used to transpose the 64 × 64-bit submatrices.

*BSM* manipulates a binary stream. The stream is first split into two streams. Then a new binary stream is computed based on those two. Finally, we identify sequences of zero runs in the stream. For each sequence identified, we output the starting position and the length of the sequence. These operations are performed with three threads (*generator*, *splitter* and *counter*). The *generator* generates the first bit stream and deposits it into a stream buffer in the scratchpad. *Splitter* splits it and deposits the two resulting streams into two stream buffers. *Splitter* also computes an intermediate stream, which is also stored in the scratchpad. The *counter* takes elements from the three stream buffers, calculates the final stream and calculate statistics on zero runs. The input stream is 1M bits.

*3DES* performs 3DES encryption in counter mode for 80k bytes. Four threads are used, each of which works on a partition of the input data.

*PartRadio* is an FM radio with multi-band equalizer. The input (10k floating point numbers) passes through a demodulator to produce an audio signal, and then an equalizer. We use three pipelined threads: a low pass filter, then a demodulator, and then an equalizer.

*Stream* [McC] is a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. The benchmark evaluates the performance of four simple vector kernels: Copy, Scale, Add and Triad. On the NMP, we run four threads in parallel, each of which processes a partition of the input data. The input parameter (memory size) is 8M.
3.5.2 Main results

Figure 3.16 shows the speedups of the applications running on the NMP over running on the main processor. Recall that the main processor has an aggressive hardware prefetcher (Section 3.5.1). In the figure, the Copy, Scale, Add and Triad bars correspond to the four components of the Stream application [McC]. The rightmost set of bars are the geometric mean of all the applications.

For each application, we show five different bars, to see the impact of the different architectural supports in the NMP. The \textit{nmp} bars correspond to the full fledged NMP architecture. \textit{novec} is the NMP without the vector hardware support. \textit{nobit} is the NMP without the bit-manipulation hardware support. \textit{nomt} is the NMP without the streaming support and running with a single thread. Note, the streaming in the NMP requires multithreading for dynamically scheduling of kernels. Finally, \textit{none} is the NMP without any support.

Focusing first on the \textit{nmp} bars, we see that these applications typically run much faster on the NMP than on an aggressive conventional processor with a hardware prefetcher. Specifically, the speedups obtained reach 18, with a geometric mean of 5.8 for the 10 bars. Since the NMP is approximately at the same distance from memory as the main processor (Table 3.6), the speedups of the NMP do not come from shorter memory latencies. Instead, they come from a better ability to hide the memory latency (and, therefore, reduce stall time) and from architectural support for several operations common in these applications.

To better understand this effect, Figure 3.17 breaks down the execution time of the applications into time that the processor is busy executing instructions (\textit{Busy}) and time that it is stalled, mostly waiting on the memory system (\textit{Idle}). The figure shows two bars for each application; the leftmost one is for the execution on the main processor, while the rightmost one is for the execution on the full-fledged NMP. For each application, the bars are normalized to the execution time on the main processor.

From the figure, we see that most of the execution time reduction of the NMP bars comes from a large reduction in the application’s stall time. This is largely due to the better architectural support in the NMP to hide memory latency. The support includes both vector instructions with long vectors and blocked multithreading. This is consistent with the work of Espasa and Valero [EV97a] that has shown that multithreading is necessary (in addition to decoupling) to improve the resource utilization of vector processors.

In addition, the busy time also typically goes down in the NMP. This is despite the fact that the NMP is a narrower issue processor, and it should take longer than the main processor to execute the same number of instructions. The busy time goes down for the NMP because of the better support in the NMP for some of the operations required by these applications. One interesting exception is 3DES, where the busy time goes up. The reason is that this application does not need the bit manipulation instructions introduced by the NMP.

Going back to Figure 3.16, we now focus on the \textit{novec} bars. They show that vector support is critical to several of
Figure 3.16: Speedup of the applications running on the NMP over running on the main processor with an aggressive hardware prefetcher. Copy, Scale, Add and Triad are the four components of the Stream application. The rightmost set of bars are the geometric mean of all the applications.
Figure 3.17: Breakdown of the execution time of the applications on the main processor (leftmost bars) and on the full-fledged NMP (rightmost bars). For each application, the bars are normalized to the execution time on the main processor.
these applications. In particular, Rgb2yuv, 3DES, and PartRadio require the vector support in the NMP to deliver any speedup at all.

The nobit bars show the importance of the support for bit manipulation. We can see that BMT and BSM heavily rely on this support. In ConvEnc, both vector and bit manipulation support are necessary to obtain good speedups — if any one is eliminated, the speedup drops substantially.

The nomt bars show that the four components of Stream (Copy, Scale, Add and Triad) need the streaming and multithreading support. If such support is eliminated, performance drops due to the limited number of in-flight memory operations (short load/store queue).

Overall, each of the three supports present in our proposed NMP is important to speed up at least some of the applications considered. Finally, if we eliminate all the three supports (none bars), the NMP is much slower than the main processor for all the applications. It is because the NMP has a low-issue in-order core.

3.6 Related work

We briefly discuss three related areas, namely processing in memory, stream architectures, and multithreaded vector architectures.

3.6.1 Processing in memory

Processing in Memory (PIM) or Intelligent Memory architectures integrate logic and DRAM in the same chip. Some of the PIM approaches [HKK+99, KHY+99, OCS98] suggest to replace main memory by PIM chips. Since the in-memory processor directly connects to the memory banks, it has a high bandwidth and low latency to main memory. Results show a significant improvement for a variety of applications. However, PIM chips require significant modification of the DRAM and have a likely high production cost.

Our NMP is different from PIM in that it does not require modifications to the DRAM chips. The NMP can be placed on the processor chip or closer to main memory.

3.6.2 Stream architectures

A stream program is organized as streams of data processed by computation kernels. A stream processor is optimized to exploit the locality and concurrency of stream programs. Imagine [KDR+01] and Merrimac [DHE+03] are two examples of the stream architecture.

Impulse[ZFP+01] expands the traditional memory hierarchy by adding address translation hardware to the memory controller. Data items whose physical DRAM addresses are not contiguous can be mapped to contiguous shadow
addresses, which is the unused physical addresses. The memory controller can compact sparse data into dense cache lines and feed the processor with a stream of data.

The NMP architecture has some of the support of a streaming architecture, but it can be argued it enables a simpler streaming compiler. The use of blocked multithreading, in particular, avoids the need for explicitly scheduling and multiplexing the kernels on the same processor, facilitates resource (processor and register) allocation, and helps better overcome variance in the execution time of different tasks.

### 3.6.3 Multithreaded vector architecture

Espasa and Valero [EV97a, EV97b] showed that multithreading can be applied to a vector processor to greatly improve the resource utilization. In their design, vector registers are part of the context of a thread. Consequently, they are saved and restored when the thread is pre-empted and re-scheduled.

In the NMP, we have explored a different design, where the vector storage is not part of a thread’s saved context. Vectors are stored in the scratchpad, which is an area shared by all threads. Not saving the registers in a context switch reduces the overhead.
Conclusions

In Chapter 2 we have proposed various ALSO patterns for frequent pattern mining. These patterns are effective and generally applicable to various implementations of frequent pattern mining algorithms. The patterns are not tied to particular implementations or applications, and can be used in other domains.

We have verified the applicability and effectiveness of these patterns in three highly optimized frequent pattern mining algorithms. Experimental results show that each of the patterns that we used is beneficial, and there is a good overall speedup of up to 2.1. Combined with previously proposed optimization strategies [GBP+05], the overall speedup could be even greater. This is quite impressive, given that we started with implementations that had already been carefully tuned. Surprisingly, the software prefetch does not give us as much as we have expected, providing a speedup of 1.3 for the best case. Although this is consistent with some of the previous research on prefetching [RS99], it is far from the speedup of up to 2.9 in some existing work [CGM01, CGMV02, CAGM04]. There are two main reasons: First, in some previous work, the speedup is evaluated for a particular execution phase, rather than the whole application run time. Second, previous evaluations on prefetching used simulators or non-commodity processors. We believe the moderate speedup for software prefetching is normal for commodity processors.

For completeness, we mentioned some other optimizations proposed in the literature; however, we did not include them in our evaluation, as these optimizations have shown effectiveness in previous work and we wanted to focus on the new patterns and those patterns that have never been applied in this domain. We believe the patterns that we have applied in the evaluation are complementary to those that have already been studied.

Our work shows that it is not only the case that one algorithm is not always the best, but also it is not always the same set of transformations that most benefits a code. The right set of transformations depends both on the input and on the system architecture. We studied the problem to select the best set of optimizations according to data inputs. We used machine learning techniques to tackle this problem and achieved good results.

We proposed in Chapter 3 a design for an engine that can support efficiently both vector and streaming applications, while providing a simpler interface than a streaming engine where all instruction scheduling is under software control. We believe this combination to be novel. We showed that this engine supports efficiently vector benchmarks, streaming benchmarks and applications requiring bit manipulations. While not demonstrated explicitly, it is also the case that
the streaming compilers for the NMP would be simpler. There was no need for a sophisticated compiler to fuse the kernels.

We expect that increases in chip density will lead to the development of heterogeneous architectures, where functions now provided by external engines, such as GPUs, will be integrated on chip. CELL is an early example of this trend. Our work shows the potential performance advantage of such an approach in an important domain.

The evaluation presented indicates that a chip that contains an NMP in addition to a regular processor can perform significantly better than a regular processor on its own. Of course, future chips could contain multiple NMPs and multiple commodity processors. While we did not compare explicitly to a commodity processor augmented with a SIMD unit, we believe that the comparison would not be very different since the main performance bottleneck is the exposed memory latency, not the ALU speed.
Appendix A

Implementations of population count function in 32-bit mode

A.1 The naive way

The following code can be found in [And].

```c
unsigned int v; // count the number of 1s in v’s binary representation
unsigned int c; // c stores the result
for (c = 0; v; v >>= 1)
{
    c += v & 1;
}
```

A.2 Popcnt by table lookup

A conversion table BitSetTable can be built for popcnt, where BitSetTable[i] = number of 1s in i. The table lookup returns the number of 1s in a byte. Below is the C++ code from [And].

```c
const unsigned char BitsSetTable256[] =
{
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
};
```
A.3 Best scalar algorithm for popcnt

The best scalar algorithm that implement a population count operation for 32-bit operands can be found in [amd05]. It implements a branchless computation of the population count. It is based on a O(log(n)) algorithm that successively groups the bits into groups of 2, 4, 8, 16, and 32, while maintaining a count of the set bits in each group.

The problem to SIMDize this algorithm is that SIMDized multiply does not work in the same way as the scalar instruction used in Step 4. In SSE, SIMDized multiplication on 32-bit values generates 64-bit result.
A.4 SIMDized popcnt

The following code is used in our implementation and can be found in [bit]. The code shown here does not include the memory alignment.

```c
unsigned bit_count_sse2(__m128i* block, __m128i* block_end)
{
    const unsigned mu1 = 0x55555555;
    const unsigned mu2 = 0x33333333;
    const unsigned mu3 = 0x0F0F0F0F;
    const unsigned mu4 = 0x0000003F;

    // Loading masks
    __m128i m1 = _mm_set_epi32 (mu1, mu1, mu1, mu1);
    __m128i m2 = _mm_set_epi32 (mu2, mu2, mu2, mu2);
    __m128i m3 = _mm_set_epi32 (mu3, mu3, mu3, mu3);
    __m128i m4 = _mm_set_epi32 (mu4, mu4, mu4, mu4);
    __m128i mcnt;
    mcnt = _mm_xor_si128(mcnt, mcnt); // cnt = 0

    while (block < block_end)
    {
        __m128i tmp1, tmp2;
        __m128i b = _mm_load_si128(block);

        // b = (b & 0x55555555) + (b >> 1 & 0x55555555);
        tmp1 = _mm_srli_epi32(b, 1); // tmp1 = (b >> 1 & 0x55555555)
        tmp1 = _mm_and_si128(tmp1, m1);
        tmp2 = _mm_and_si128(b, m1); // tmp2 = (b & 0x55555555)
        b = _mm_add_epi32(tmp1, tmp2); // b = tmp1 + tmp2

        // b = (b & 0x33333333) + (b >> 2 & 0x33333333);
        tmp1 = _mm_srli_epi32(b, 2); // (b >> 2 & 0x33333333)
```
tmp1 = _mm_and_si128(tmp1, m2);
tmp2 = _mm_and_si128(b, m2);  // (b & 0x33333333)
b = _mm_add_epi32(tmp1, tmp2);// b = tmp1 + tmp2

// b = (b + (b >> 4)) & 0x0F0F0F0F;
tmp1 = _mm_srl_epi32(b, 4);  // tmp1 = b >> 4
b = _mm_add_epi32(b, tmp1);  // b = b + (b >> 4)
b = _mm_and_si128(b, m3);  // & 0x0F0F0F0F

// b = b + (b >> 8);
tmp1 = _mm_srl_epi32(b, 8);  // tmp1 = b >> 8
b = _mm_add_epi32(b, tmp1);  // b = b + (b >> 8)

// b = (b + (b >> 16)) & 0x0000003F;
tmp1 = _mm_srl_epi32(b, 16);  // b >> 16
b = _mm_add_epi32(b, tmp1);  // b + (b >> 16)
b = _mm_and_si128(b, m4);  // (b >> 16) & 0x0000003F;

mcnt = _mm_add_epi32(mcnt, b);  // mcnt += b

++block;
}

bm::id_t tcnt[4];
_mm_storeu_si128((__m128i*)tcnt, mcnt);

}
References


[AMD00b] AMD. AMD extensions to the 3dnow!™ and MMX™ instruction sets. Publication 22466D, March 2000.


[cra]     Cray assembly language (CAL) for Cray X1 system reference manual.


[Han]     Pat Hanrahan. private communication.


[LPWH02] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining frequent item sets by opportunistic projection. In KDD’02, pages 229–238, Edmonton, Alberta, Canada, 2002. ACM Press.


Author’s Biography

Mingliang Wei was born in 1976 and spent his childhood in Xuzhou, Jiangsu Province, P.R.China. He moved with his parents to Jinzhou, Liaoning Province in 1985. He earned his B.S. and M.E. degrees in computer science from Nanjing University, China. He spent one year at Rensselaer Polytechnic Institute before he finally transferred to the University of Illinois at Urbana-Champaign as a Ph.D. student in computer science.

He joined the P3 (Parallel Processing Principles) group at UIUC and worked with Prof. Marc Snir and Prof. Josep Torrellas on the PERCS (Productive, Easy-to-use, Reliable Computing System) project, in which he designed a near-memory processor that is optimized for vector, streaming and bit-manipulation tasks. In Summer 2004, he worked on the PERCS project at IBM T. J. Watson Research Center in Yorktown, NY. During his Ph.D. study, he also worked on performance tuning and algorithm selection for frequent pattern mining.